JFP **34**, e16, 119 pages, 2025.

doi:10.1017/S0956796824000133

From high to low: Simulating nondeterminism and state with state

WENHAO TANG
The University of Edinburgh, Edinburgh, UK
(*e-mail:* wenhao.tang@ed.ac.uk)

TOM SCHRIJVERS
Department of Computer Science, KU Leuven, Leuven, Belgium
(*e-mail:* tom.schrijvers@kuleuven.be)

Abstract

Some effects are considered to be higher level than others. High-level effects provide expressive and succinct abstraction of programming concepts, while low-level effects allow more fine-grained control over program execution and resources. Yet, often it is desirable to write programs using the convenient abstraction offered by high-level effects, and meanwhile still benefit from the optimizations enabled by low-level effects. One solution is to translate high-level effects to low-level ones.

This paper studies how algebraic effects and handlers allow us to simulate high-level effects in terms of low-level effects. In particular, we focus on the interaction between state and nondeterminism known as the local state, as provided by Prolog. We map this high-level semantics in successive steps onto a low-level composite state effect, similar to that managed by Prolog's Warren Abstract Machine. We first give a translation from the high-level local-state semantics to the low-level global-state semantics, by explicitly restoring state updates on backtracking. Next, we eliminate nondeterminism altogether in favour of a lower-level state containing a choicepoint stack. Then we avoid copying the state by restricting ourselves to incremental, reversible state updates. We show how these updates can be stored on a trail stack with another state effect. We prove the correctness of all our steps using program calculation where the fusion laws of effect handlers play a central role.

1 Introduction

The trade-off between "high-level" and "low-level" styles of programming is almost as old as the field of computer science itself. In a high-level style of programming, we lean on abstractions to make our programs easier to read and write and less error prone. We pay for this comfort by giving up precise control over the underlying machinery; we forego optimization opportunities or have to trust a (usually opaque) compiler to perform low-level optimizations for us. For performance-sensitive applications, compiler optimizations are not reliable enough; instead we often resort to lower-level programming techniques ourselves. Although these lower-level programming techniques allow a fine-grained control over program execution and the implementation of optimization techniques, they tend

to be harder to write and not compose very well. This is an important trade-off to take into account when choosing an appropriate programming language for implementing an application.

Maybe surprisingly, as they are rarely described in this way, there is a similar pattern for side effects within programming languages: some effects can be described as "lower-level" than others. Informally, we say that an effect is lower-level than another effect when the lower-level effect can simulate the higher-level effect. In other words, it is possible to write a program using lower-level effects that has identical semantics to the same program with higher-level effects. Yet, due to the lack of abstraction of low-level effects, writing a faithful simulation requires careful discipline and is quite error prone.

This article investigates how we can construct programs that are most naturally expressed with a high-level effect, but where we still want access to the optimization opportunities of a lower-level effect. In particular, inspired by Prolog and Constraint Programming systems, we investigate programs that rely on high-level interaction between the nondeterminism and state effects which we call *local state*. Following low-level implementation techniques for these systems, like the Warren Abstract Machine (WAM) (Warren, 1983; Aït-Kaci, 1991), we show how these high-level effects can be simulated in terms of the low-level *global state* interaction of state and nondeterminism, and finally by state alone. This allows us to incorporate typical optimizations like exploiting mutable state for efficient backtracking based on *trailing* as opposed to copying or recomputing the state from scratch (Schulte, 1999).

Our approach is based on algebraic effects and handlers (Plotkin and Power, 2003; Plotkin and Pretnar, 2009, 2013) to cleanly separate the syntax and semantics of effects. For programs written with high-level effects and interpreted by their handlers, we can define a general translation handler to transform these high-level effects to low-level effects and then interpret the translated programs with the handlers of low-level effects. Though we do not give a formal definition of a simulation from high-level effects to low-level effects, we expect it to be a handler that interprets the operations of the high-level effects in terms of the operations of the low-level effects. This handler is essentially a monomorphism from the syntax tree of high-level effects to that of low-level effects, similar to Felleisen (1991)'s notion of macro expansion but at the continuation-passing level (as the syntax trees of free monads provide access to continuations).

Of particular interest is the way we reason about the correctness of our approach. There has been much debate in the literature on different equational reasoning approaches for effectful computations. Hutton and Fulger (2008) break the abstraction boundaries and use the actual implementation in their equational reasoning approach. Gibbons and Hinze (2011) promote an alternative, law-based approach to preserve abstraction boundaries and combine axiomatic with equational reasoning. In an earlier version of this work (Pauwels et al., 2019), we have followed the latter, law-based approach for reasoning about the correctness of simulating local state with global state. However, we have found that approach to be unsatisfactory because it incorporates elements that are usually found in the syntactic approach for reasoning about programming languages (Wright and Felleisen, 1994), leading to more boilerplate and complication in the proofs: notions of contextual equivalence and explicit manipulation of program contexts. Hence, for that reason we return to the implementation-based reasoning approach, which we believe works well with algebraic

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Table 1: Overview of translations from high-level effects to low-level effects in the paper

Translations	Descriptions	Correctness
local2global	Local state to global state (Section 4.3)	Theorem 1
nondet2state	Nondeterminism to state (Section 5.2)	Theorem 3
states2state	Multiple states to a single state (Section 6.1)	Theorem 4
$local2global_M$	Local state to global state with reversible updates (Section 7.3)	Theorem 6
local2trail	Local state to global state with trail stacks (Section 8.1)	Theorem 7

effects and handlers. Indeed, we prove all of our simulations correct using equational reasoning techniques, exploiting in particular the fusion property of handlers (Wu and Schrijvers, 2015; Gibbons, 2000).

After introducing the reader to the appropriate background material and motivating the problem (Sections 2 and 3), this paper makes the following contributions:

- We distinguish between local-state and global-state semantics and simulate the former in terms of the latter (Section 4).
- We simulate nondeterminism using a state that consists of a choicepoint stack (Section 5).
- We combine the previous two simulations and merge the two states into a single state effect (Section 6).
- By only allowing incremental, reversible updates to the state we can avoid holding on to multiple copies of the state (Section 7).
- By storing the incremental updates in a trail stack state, we can restore them in batch when backtracking (Section 8).
- We prove all simulations correct using equational reasoning techniques and the fusion law for handlers in particular (Appendices 2, 3, 4, 5, 6, and 7).

Finally, we discuss related work (Section 9) and conclude (Section 10). Table 1 gives an overview of the simulations of high-level effects with low-level effects, we implemented and proved in the paper. Throughout the paper, we use Haskell as a means to illustrate our findings with code. In particular, we restrict ourselves to a well-behaved and well-founded fragment of Haskell that avoids non-termination and other forms of bottom and readily admits equational reasoning with structural induction. Moreover, we focus on formalising and proving the correctness of the simulations rather than empirical evidence of performance improvements, as those have already been demonstrated by real-world systems like Prolog. In fact, the Haskell implementations themselves do not exhibit performance improvements due to aspects like laziness, immutable state, and the overhead of algebraic effects and handlers.

2 Background and motivation

This section summarizes the main prerequisites for equational reasoning with effects and motivates our translations from high-level effects to low-level effects. We discuss the two central effects of this paper: state and nondeterminism.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

2.1 Functors and monads

Functors. In Haskell, a functor $f :: * \to *$ instantiates the functor type class, which has a single functor mapping operation.

class *Functor f* **where**
  $fmap :: (a \to b) \to f\ a \to f\ b$

Furthermore, a functor should satisfy the following two functor laws:

$$\textbf{identity}: \quad fmap\ id = id, \tag{2.1}$$

$$\textbf{composition}: \quad fmap\ (f \circ g) = fmap\ f \circ fmap\ g. \tag{2.2}$$

We sometimes use the operator $\langle\$\rangle$ as an alias for *fmap*.

$(\langle\$\rangle) :: Functor\ f \Rightarrow (a \to b) \to f\ a \to f\ b$
$(\langle\$\rangle) = fmap$

Monads. Monadic side effects (Moggi, 1991), the main focus of this paper, are those that can dynamically determine what happens next. A monad $m :: * \to *$ is a functor which instantiates the monad type class, which has two operations return (η) and bind ($\gg\!=$).

class *Functor m* $\Rightarrow$ *Monad m* **where**
  $\eta \quad :: a \to m\ a$
  $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$

Furthermore, a monad should satisfy the following three monad laws:

$$\textbf{return-bind}: \quad \eta\ x \gg\!= f = f\ x, \tag{2.3}$$

$$\textbf{bind-return}: \quad m \gg\!= \eta = m, \tag{2.4}$$

$$\textbf{associativity}: \quad (m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \to f\ x \gg\!= g). \tag{2.5}$$

Haskell supports **do** blocks as syntactic sugar for monadic computations. For example, **do** $x \leftarrow m; f\ x$ is translated to $m \gg\!= f$. Two convenient derived operators are $\gg$ and $\langle * \rangle$.[1]

$(\gg) :: Monad\ m \Rightarrow m\ a \to m\ b \to m\ b$
$m_1 \gg m_2 = m_1 \gg\!= \lambda\_ \to m_2$
$(\langle * \rangle) :: Monad\ m \Rightarrow m\ (a \to b) \to m\ a \to m\ b$
$mf\ \langle * \rangle\ mx = mf \gg\!= \lambda f \to mx \gg\!= \lambda x \to \eta\ (f\ x)$

2.2 Nondeterminism and state

Following both the approaches of Hutton and Fulger (2008) and of Gibbons and Hinze (2011), we introduce effects as subclasses of the *Monad* type class.

Nondeterminism. The first monadic effect we introduce is nondeterminism. We define a subclass *MNondet* of *Monad* to capture the nondeterministic interfaces as follows:

[1] We deviate from the type class hierarchy of *Functor*, *Applicative*, and *Monad* that can be found in Haskell's standard library because its additional complexity is not needed in this article.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

```
class Monad m ⇒ MNondet m where
  ∅  :: m a
  ([]) :: m a → m a → m a
```

Here, $\varnothing$ denotes failures and $([])$ denotes nondeterministic choices. Instances of the *MNondet* interface should satisfy the following four laws: [2]

$$\textbf{identity}: \quad \varnothing \,[]\, m = m = m \,[]\, \varnothing, \tag{2.6}$$

$$\textbf{associativity}: \quad (m \,[]\, n) \,[]\, k = m \,[]\, (n \,[]\, k), \tag{2.7}$$

$$\textbf{right-distributivity}: \quad (m_1 \,[]\, m_2) \gg\!= f = (m_1 \gg\!= f) \,[]\, (m_2 \gg\!= f), \tag{2.8}$$

$$\textbf{left-identity}: \quad \varnothing \gg\!= f = \varnothing. \tag{2.9}$$

The first two laws state that $([])$ and $\varnothing$ should form a monoid, i.e., $([])$ should be associative with $\varnothing$ as its neutral element. The last two laws show that $(\gg\!=)$ is right-distributive over $([])$ and that $\varnothing$ cancels bind on the left.

The approach of Gibbons and Hinze (2011) is to reason about effectful programs using an axiomatic characterization given by these laws. It does not rely on the specific implementation of any particular instance of *MNondet*. In contrast, Hutton and Fulger (2008) reason directly in terms of a particular instance. In the case of *MNondet*, the quintessential instance is the list monad, which extends the conventional *Monad* instance for lists.

```
instance MNondet [ ] where
  ∅  = [ ]
  ([]) = (++)

instance Monad [ ] where
  η x = [x]
  xs ≫= f = concatMap f xs
```

State. The signature for the state effect has two operations: a *get* operation that reads and returns the state and a *put* operation that modifies the state, overwriting it with the given value, and returns nothing. Again, we define a subclass *MState* of *Monad* to capture its interfaces.

```
class Monad m ⇒ MState s m | m → s where
  get :: m s
  put :: s → m ()
```

These operations are regulated by the following four laws:

$$\textbf{put-put}: \quad put\ s \gg put\ s' = put\ s', \tag{2.10}$$

$$\textbf{put-get}: \quad put\ s \gg get = put\ s \gg \eta\ s, \tag{2.11}$$

$$\textbf{get-put}: \quad get \gg\!= put = \eta\ (), \tag{2.12}$$

$$\textbf{get-get}: \quad get \gg\!= (\lambda s \to get \gg\!= k\ s) = get \gg\!= (\lambda s \to k\ s\ s). \tag{2.13}$$

[2] One might expect additional laws such as idempotence or commutativity. As argued by Kiselyov (2015), these laws differ depending on how the monad is used and how it should interact with other effects. The standard *MonadPlus* type class has no laws associated. We introduce a different type class, *MNondet*, to impose the minimal set of laws for nondeterminism from Rivas et al. (2018). We choose these laws because they are consistent with both the list monad and with the behaviour of Prolog.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

The standard instance of *MState* is the state monad *State s*.

```
newtype State s a =
  State {run_State :: s → (a, s)}
instance MState s (State s) where
  get   = State (λs → (s, s))
  put s = State (λ_ → ((), s))

instance Monad (State s) where
  η x = State (λs → (x, s))
  m >>= f = State (λs → let (x, s') = run_State m s
                        in run_State (f x) s')
```

2.3 The N-queens puzzle

The n-queens problem used here is an adapted and simplified version from that of Gibbons and Hinze (2011). The aim of the puzzle is to place n queens on a $n \times n$ chess board such that no two queens can attack each other. This means that no two queens should be placed on the same row, the same column or the same diagonal of the chess board.

Given n, we number the rows and columns by $[1\,..\,n]$. 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 $[1\,..\,n]$, such that $xs\,!!\,i = j$ denotes that the queen on the ith column is placed on the jth row. Using this representation, queens cannot be put on the same row or column.

A naive algorithm. We have the following naive nondeterministic algorithm for n-queens.

```
queens_naive :: MNondet m ⇒ Int → m [Int]
queens_naive n = choose (permutations [1 .. n]) >>= filtr valid
```

The program $queens_{naive}\ 4 :: [[Int]]$ gives as result $[[2,4,1,3],[3,1,4,2]]$. The program uses a generate-and-test strategy: it generates all permutations of queens as candidate solutions, and then tests which ones are valid.

The function *permutations* :: $[a] \to [[a]]$ from *Data.List* computes all the permutations of its input. The function *choose* implemented as follows nondeterministically picks an element from a list.

```
choose :: MNondet m ⇒ [a] → m a
choose = foldr ((▯) ∘ η) ∅
```

The function *filtr p x* returns x if $p\,x$ holds and fails otherwise.

```
filtr :: MNondet m ⇒ (a → Bool) → a → m a
filtr p x = if p x then η x else ∅
```

The pure function *valid* :: $[Int] \to Bool$ determines whether the input is a valid solution.

```
valid :: [Int] → Bool
valid []       = True
valid (q : qs) = valid qs ∧ safe q 1 qs
```

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

A solution is valid when each queen is *safe* with respect to the subsequent queens:

$$
\begin{aligned}
&\mathit{safe} :: \mathit{Int} \to \mathit{Int} \to [\mathit{Int}] \to \mathit{Bool}\\
&\mathit{safe}\ \_\ \_\ [\,] \qquad = \mathit{True}\\
&\mathit{safe}\ q\ n\ (q_1 : qs) = \mathit{and}\ [q \not\equiv q_1, q \not\equiv q_1 + n, q \not\equiv q_1 - n, \mathit{safe}\ q\ (n+1)\ qs]
\end{aligned}
$$

The call $\mathit{safe}\ q\ n\ qs$ checks whether the current queen q is on a different ascending and descending diagonal than the other queens qs, where n is the number of columns that q is apart from the first queen q_1 in qs.

Although this generate-and-test approach works and is quite intuitive, it is not very efficient. For example, all solutions of the form $(1 : 2 : qs)$ are invalid because the first two queens are on the same diagonal. However, the algorithm still needs to generate and test all $(n-2)!$ candidate solutions of this form.

A backtracking algorithm. We can fuse the two phases of the naive algorithm to obtain a more efficient algorithm, where both generating candidates and checking for validity happens in a single pass. The idea is to move to a state-based backtracking implementation that allows early pruning of branches that are invalid. In particular, when placing the new queen in the next column, we make sure that it is only placed in positions that are valid with respect to the previously placed queens.

We use a state $(\mathit{Int}, [\mathit{Int}])$ to contain the current column and the previously placed queens. The backtracking algorithm of n-queens is implemented as follows.

$$
\begin{aligned}
&\mathit{queens} :: (\mathit{MState}\ (\mathit{Int}, [\mathit{Int}])\ m, \mathit{MNondet}\ m) \Rightarrow \mathit{Int} \to m\ [\mathit{Int}]\\
&\mathit{queens}\ n = \mathit{loop}\ \mathbf{where}\\
&\quad \mathit{loop} = \mathbf{do}\ (c, sol) \leftarrow \mathit{get}\\
&\qquad\qquad \mathbf{if}\ c \geqslant n\ \mathbf{then}\ \eta\ sol\\
&\qquad\qquad \mathbf{else\ do}\ r \leftarrow \mathit{choose}\ [1\,..\,n]\\
&\qquad\qquad\qquad\quad \mathit{guard}\ (\mathit{safe}\ r\ 1\ sol)\\
&\qquad\qquad\qquad\quad s \leftarrow \mathit{get}\\
&\qquad\qquad\qquad\quad \mathit{put}\ (s \oplus r)\\
&\qquad\qquad\qquad\quad \mathit{loop}
\end{aligned}
$$

The function *guard* fails when the input is false.

$$
\begin{aligned}
&\mathit{guard} :: \mathit{MNondet}\ m \Rightarrow \mathit{Bool} \to m\ ()\\
&\mathit{guard}\ \mathit{True} = \eta\ ()\\
&\mathit{guard}\ \mathit{False} = \varnothing
\end{aligned}
$$

The function $s \oplus r$ updates the state with a new queen placed on row r in the next column.

$$
\begin{aligned}
&(\oplus)\ :: (\mathit{Int}, [\mathit{Int}]) \to \mathit{Int} \to (\mathit{Int}, [\mathit{Int}])\\
&(\oplus)\ (c, sol)\ r = (c+1, r : sol)
\end{aligned}
$$

The above monadic version of *queens* essentially assume that each searching branch has its own state; we do not need to explicitly restore the state when backtracking. Though it is a convenient high-level programming assumption for programmers, it causes obstacles to low-level implementations and optimizations. In the following sections, we investigate how low-level implementation and optimization techniques, such as those

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

found in Prolog's Warren Abstract Machine and Constraint Programming systems, can be incorporated and proved correct.

3 Algebraic effects and handlers

This section introduces algebraic effects and handlers, the approach we use to define syntax, semantics, and simulations for effects. Comparing to giving concrete monad implementations for effects, algebraic effects and handlers allow us to easily provide different interpretations for the same effects due to the clear separation of syntax and semantics. As a result, we can smoothly specify translations from high-level effects to low-level effects as handlers of these high-level effects and then compose them with the handlers of low-level effects to interpret high-level programs. Algebraic effects and handlers also provide us with a modular way to combine our translations with other effects, and a useful tool, the fusion property, to prove the correctness of translations.

3.1 Free monads and their folds

We implement algebraic effects and handlers as free monads and their folds.

Free monads. Free monads are gaining popularity for their use in algebraic effects (Plotkin and Power, 2002, 2003) and handlers (Plotkin and Pretnar, 2009, 2013), which elegantly separate syntax and semantics of effectful operations. A free monad, the syntax of an effectful program, can be captured generically in Haskell.

$$\textbf{data}\ \mathit{Free}\ f\ a = \mathit{Var}\ a \mid \mathit{Op}\ (f\ (\mathit{Free}\ f\ a))$$

This data type is a form of abstract syntax tree (AST) consisting of leaves ($\mathit{Var}\ a$) and internal nodes ($\mathit{Op}\ (f\ (\mathit{Free}\ f\ a))$), whose branching structure is determined by the functor f. This functor is also known as the *signature* of operations.

A fold recursion scheme. Free monads come equipped with a fold recursion scheme.

$$\begin{aligned}&\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (a \to b) \to (f\ b \to b) \to \mathit{Free}\ f\ a \to b\\ &\mathit{fold}\ \mathit{gen}\ \mathit{alg}\ (\mathit{Var}\ x) \ = \mathit{gen}\ x\\ &\mathit{fold}\ \mathit{gen}\ \mathit{alg}\ (\mathit{Op}\ \mathit{op}) = \mathit{alg}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{gen}\ \mathit{alg})\ \mathit{op})\end{aligned}$$

This fold interprets an AST structure of type $\mathit{Free}\ f\ a$ into some semantic domain b. It does so compositionally using a generator $\mathit{gen} :: a \to b$ for the leaves and an algebra $\mathit{alg} :: f\ b \to b$ for the internal nodes; together these are also known as a *handler*.

The monad instance of *Free* is straightforwardly implemented with fold.

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{Free}\ f)\ \textbf{where}\\ &\quad \eta \quad\ \ = \mathit{Var}\\ &\quad m \gg\!= f = \mathit{fold}\ f\ \mathit{Op}\ m\end{aligned}$$

Under certain conditions folds can be fused with functions that are composed with them (Wu and Schrijvers, 2015; Gibbons, 2000). This gives rise to the following laws:

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$\textbf{fusion-pre}: \quad fold\ (gen \circ h)\ alg = fold\ gen\ alg \circ fmap\ h, \tag{3.1}$$

$$\textbf{fusion-post}: \quad h \circ fold\ gen\ alg = fold\ (h \circ gen)\ alg' \text{ with } h \circ alg = alg' \circ fmap\ h, \tag{3.2}$$

$$\textbf{fusion-post'}: \quad h \circ fold\ gen\ alg = fold\ (h \circ gen)\ alg' \text{ with } h \circ alg \circ fmap\ f = alg' \circ fmap\ h \circ fmap\ f \text{ and } f = fold\ gen\ alg. \tag{3.3}$$

These three fusion laws turn out to be essential in the further proofs of this paper.

Nondeterminism. Instead of using a concrete monad like *List*, we use the free monad $Free\ Nondet_F$ over the signature $Nondet_F$ following algebraic effects.

$$\textbf{data}\ Nondet_F\ a = Fail \mid Or\ a\ a$$

This signatures gives rise to a trivial *MNondet* instance:

$$\begin{aligned}&\textbf{instance}\ MNondet\ (Free\ Nondet_F)\ \textbf{where}\\ &\quad \varnothing \quad\ = Op\ Fail\\ &\quad (\llbracket)\ p\ q = Op\ (Or\ p\ q)\end{aligned}$$

With this representation the **right-distributivity** law and the **left-identity** law follow trivially from the definition of $(\gg\!=)$ for the free monad.

In contrast, the **identity** and **associativity** laws are not satisfied on the nose. Indeed, $Op\ (Or\ Fail\ p)$ is for instance a different abstract syntax tree than p. Yet, these syntactic differences do not matter as long as their interpretation is the same. This is where the handlers come in; the meaning they assign to effectful programs should respect the laws. We have the following h_{ND} handler which interprets the free monad in terms of lists.

$$\begin{aligned}&h_{ND} :: Free\ Nondet_F\ a \to [a]\\ &h_{ND} = fold\ gen_{ND}\ alg_{ND}\\ &\quad \textbf{where}\\ &\qquad gen_{ND}\ x \quad\ = [x]\\ &\qquad alg_{ND}\ Fail \quad = [\,]\\ &\qquad alg_{ND}\ (Or\ p\ q) = p +\!\!+ q\end{aligned}$$

With this handler, the **identity** and **associativity** laws are satisfied up to handling as follows:

$$\begin{aligned}h_{ND}\ (\varnothing \llbracket m) &= h_{ND}\ m = h_{ND}\ (m \llbracket \varnothing)\\ h_{ND}\ ((m \llbracket n) \llbracket o) &= h_{ND}\ (m \llbracket (n \llbracket o))\end{aligned}$$

In fact, two stronger *contextual* equalities hold:

$$\begin{aligned}h_{ND}\ ((\varnothing \llbracket m) \gg\!= k) &= h_{ND}\ (m \gg\!= k) = h_{ND}\ ((m \llbracket \varnothing) \gg\!= k)\\ h_{ND}\ (((m \llbracket n) \llbracket o) \gg\!= k) &= h_{ND}\ ((m \llbracket (n \llbracket o)) \gg\!= k)\end{aligned}$$

These equations state that the interpretations of the left- and right-hand sides are indistinguishable even when put in a larger program context $\gg\!= k$. They follow from the definitions of h_{ND} and $(\gg\!=)$, as well as the associativity and identity properties of $(+\!\!+)$.

We obtain the two non-contextual equations as a corollary by choosing $k = \eta$.

State. Again, instead of using the concrete *State* monad in Section 2.2, we model states via the free monad $Free\ (State_F\ s)$ over the state signature.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$\textbf{data}\ State_F\ s\ a = Get\ (s \to a) \mid Put\ s\ a$$

This state signature gives the following *MState s* instance:

$$\begin{aligned}&\textbf{instance}\ MState\ s\ (Free\ (State_F\ s))\ \textbf{where}\\ &\quad get\ \ = Op\ (Get\ \eta)\\ &\quad put\ s = Op\ (Put\ s\ (\eta\ ()))\end{aligned}$$

The following handler h'_{State} maps this free monad to the *State s* monad.

$$\begin{aligned}&h'_{State} :: Free\ (State_F\ s)\ a \to State\ s\ a\\ &h'_{State} = fold\ gen'_S\ alg'_S\\ &\quad \textbf{where}\\ &\qquad gen'_S\ x\ \ \ \ \ \ \ \ \ \ = State\ \$\ \lambda s \to (x, s)\\ &\qquad alg'_S\ (Get\ \ \ k) = State\ \$\ \lambda s \to run_{State}\ (k\ s)\ s\\ &\qquad alg'_S\ (Put\ s'\ k) = State\ \$\ \lambda s \to run_{State}\ k\ s'\end{aligned}$$

It is easy to verify that the four state laws hold contextually up to interpretation with h'_{State}.

3.2 Modularly combining effects

Combining multiple effects is relatively easy in the axiomatic approach based on type classes. By imposing multiple constraints on the monad *m*, e.g. (*MState s m*, *MNondet m*), we can express that *m* should support both state and nondeterminism and respect their associated laws. In practice, this is often insufficient: we usually require additional laws that govern the interactions between the combined effects. We discuss possible interaction laws between state and nondeterminism in details in Section 4.

The coproduct operator for combining effects. To combine the syntax of effects given by free monads, we need to define a right-associative coproduct operator :+: for signatures.

$$\textbf{data}\ (f :+: g)\ a = Inl\ (f\ a) \mid Inr\ (g\ a)$$

Note that given two functors *f* and *g*, it is obvious that $f :+: g$ is again a functor. This coproduct operator allows a modular definition of the signatures of combined effects. For instance, we can encode programs with both state and nondeterminism as effects using the data type $Free\ (State_F :+: Nondet_F)\ a$. The coproduct also has a neutral element Nil_F, representing the empty effect set.

$$\textbf{data}\ Nil_F\ a \quad \text{-- no constructors}$$

We define the following two instances, which allow us to compose state effects with any other effect functor *f*, and nondeterminism effects with any other effect functors *f* and *g*, respectively. As a result, it is easy to see that $Free\ (State_F\ s :+: Nondet_F :+: f)$ supports both state and nondeterminism for any functor *f*.

$$\begin{aligned}&\textbf{instance}\ (Functor\ f) \Rightarrow MState\ s\ (Free\ (State_F\ s :+: f))\ \textbf{where}\\ &\quad get\ \ = Op\ \$\ Inl\ \$\ Get\ \eta\\ &\quad put\ x = Op\ \$\ Inl\ \$\ Put\ x\ (\eta\ ())\\ &\textbf{instance}\ (Functor\ f, Functor\ g) \Rightarrow MNondet\ (Free\ (g :+: Nondet_F :+: f))\ \textbf{where}\end{aligned}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$\begin{aligned}\varnothing &= Op\ \$\ Inr\ \$\ Inl\ Fail\\ x\ [\!]\ y &= Op\ \$\ Inr\ \$\ Inl\ (Or\ x\ y)\end{aligned}$$

Modularly combining effect handlers. In order to interpret composite signatures, we use the forwarding approach of Schrijvers et al. (2019). This way the handlers can be modularly composed: they only need to know about the part of the syntax their effect is handling and forward the rest of the syntax to other handlers.

A mediator (#) is used to separate the algebra *alg* for the handled effects and the forwarding algebra *fwd* for the unhandled effects.

$$\begin{aligned}&(\#) :: (f\ a \to b) \to (g\ a \to b) \to (f :+: g)\ a \to b\\ &(alg\ \#\ fwd)\ (Inl\ op) = alg\ op\\ &(alg\ \#\ fwd)\ (Inr\ op) = fwd\ op\end{aligned}$$

The handlers for state and nondeterminism we have given earlier require a bit of adjustment to be used in the composite setting since they only consider the signature of their own effects. We need to interpret the free monads into composite domains, $StateT\ (Free\ f)\ a$ and $Free\ f\ [a]$, respectively. Here, *StateT* is the state transformer from the Monad Transformer Library (Jones, 1995).

$$\textbf{newtype}\ StateT\ s\ m\ a = StateT\ \{run_{StateT} :: s \to m\ (a, s)\}$$

The new handlers, into these composite domains, are defined as follows:

$$\begin{aligned}&h_{State} :: Functor\ f \Rightarrow Free\ (State_F\ s :+: f)\ a \to StateT\ s\ (Free\ f)\ a\\ &h_{State} = fold\ gen_S\ (alg_S\ \#\ fwd_S)\\ &\quad \textbf{where}\\ &\qquad gen_S\ x \qquad\quad = StateT\ \$\ \lambda s \to \eta\ (x, s)\\ &\qquad alg_S\ (Get\ \ \ k) = StateT\ \$\ \lambda s \to run_{StateT}\ (k\ s)\ s\\ &\qquad alg_S\ (Put\ s'\ k) = StateT\ \$\ \lambda s \to run_{StateT}\ k\ s'\\ &\qquad fwd_S\ op \qquad = StateT\ \$\ \lambda s \to Op\ \$\ fmap\ (\lambda k \to run_{StateT}\ k\ s)\ op\end{aligned}$$

$$\begin{aligned}&h_{ND+f} :: Functor\ f \Rightarrow Free\ (Nondet_F :+: f)\ a \to Free\ f\ [a]\\ &h_{ND+f} = fold\ gen_{ND+f}\ (alg_{ND+f}\ \#\ fwd_{ND+f})\\ &\quad \textbf{where}\\ &\qquad gen_{ND+f} \qquad = Var \circ \eta\\ &\qquad alg_{ND+f}\ Fail \qquad = Var\ [\,]\\ &\qquad alg_{ND+f}\ (Or\ p\ q) = liftM2\ (+\!+)\ p\ q\\ &\qquad fwd_{ND+f}\ op \qquad = Op\ op\end{aligned}$$

Also, the empty signature Nil_F has a trivial associated handler.

$$\begin{aligned}&h_{Nil} :: Free\ Nil_F\ a \to a\\ &h_{Nil}\ (Var\ x) = x\end{aligned}$$

3.3 Proof device

The algebraic effects and handlers implementation we introduce in this paper is, much like the core calculus of a programming language, a proof device and not an ergonomic library. The results we obtain for this representation can be transferred to other representations.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Explicit isomorphisms. For instance, notice for instance that h_{State} and h_{ND+f} both require the signature they handle to be on the left in the co-product. It is possible to relax this requirement by means of advanced type-level programming, e.g., using type class overloading (Swierstra, 2008). That makes using handlers more ergonomic at the cost of obscuring formal reasoning about them. Because the latter is the focus of this paper, we do not introduce the additional flexibility. Instead, we appeal to explicit isomorphisms, to reorder the signatures in a co-product. For example, the $(\Leftrightarrow)$ isomorphism that we will use in Section 4.2 swaps the order of two functors in the co-product signature of the free monad.

$$
\begin{aligned}
&(\Leftrightarrow) :: (\mathit{Functor}\, f_1, \mathit{Functor}\, f_2, \mathit{Functor}\, f) \Rightarrow \mathit{Free}\, (f_1 :+: f_2 :+: f)\, a \to \mathit{Free}\, (f_2 :+: f_1 :+: f)\, a \\
&(\Leftrightarrow)\, (\mathit{Var}\, x) \qquad\quad = \mathit{Var}\, x \\
&(\Leftrightarrow)\, (\mathit{Op}\, (\mathit{Inl}\, k)) \qquad = (\mathit{Op} \circ \mathit{Inr} \circ \mathit{Inl})\ (\mathit{fmap}\, (\Leftrightarrow)\, k) \\
&(\Leftrightarrow)\, (\mathit{Op}\, (\mathit{Inr}\, (\mathit{Inl}\, k))) = (\mathit{Op} \circ \mathit{Inl}) \qquad (\mathit{fmap}\, (\Leftrightarrow)\, k) \\
&(\Leftrightarrow)\, (\mathit{Op}\, (\mathit{Inr}\, (\mathit{Inr}\, k))) = (\mathit{Op} \circ \mathit{Inr} \circ \mathit{Inr})\ (\mathit{fmap}\, (\Leftrightarrow)\, k)
\end{aligned}
$$

Transfer to other representations. Our use of type class constraints allows us to reduce other monadic representations to the core algebraic effects and handlers representation by an appeal to parametricity (Voigtländer, 2009). For instance, for a program $p :: \forall m.\mathit{MNondet}\, m \Rightarrow m\, \mathit{Int}$ we have that:

$$p :: [\mathit{Int}] = h_{ND}\, (p :: \mathit{Free}\, \mathit{Nondet}_F\, \mathit{Int})$$

This is true because h_{ND} is the structure-preserving map from the $\mathit{Free}\, \mathit{Nondet}_F$ instance of *MNondet* to the [ ] instance. That is to say, h_{ND} satisfies the following four equations:

$$
\begin{aligned}
h_{ND}\, (\eta\, x) &= \eta\, x \\
h_{ND}\, (m \gg\!= k) &= h_{ND}\, m \gg\!= h_{ND} \circ k \\
h_{ND}\, \varnothing &= \varnothing \\
h_{ND}\, (m \,[\!]\, n) &= h_{ND}\, m \,[\!]\, h_{ND}\, n
\end{aligned}
$$

Since the free-monad representation is initial, the structure-preserving map h_{ND} is guaranteed to exist and be unique. Now, if we want to prove a property about $p :: [\mathit{Int}]$, the parametricity equation allows us to prove it instead about $h_{ND}\, (p :: \mathit{Free}\, \mathit{Nondet}_F\, \mathit{Int})$. A similar observation can be made for other constraints, like *MState* or the combination of *MState* and *MNondet*.

In the rest of this paper, we focus on results for the core representation of algebraic effects and handlers. By means of the above approach, these results can be generalized to other representations.

4 Modelling local state with global state

This section studies two flavours of effect interaction between state and nondeterminism: local-state and global-state semantics. Local state is a higher-level effect than global state. In a program with local state, each nondeterministic branch has its own local copy of the state. This is a convenient programming abstraction provided by many systems that solve search problems, e.g., Prolog. In contrast, global state linearly threads a single state

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

through the nondeterministic branches. This can be interesting for performance reasons: we can limit memory use by avoiding multiple copies, and perform in-place updates to reduce allocation and garbage collection, and to improve locality.

In this section, we first formally characterize local-state and global-state semantics, and then define a translation from the former to the latter which uses the mechanism of nondeterminism to store previous states and insert backtracking branches.

4.1 Local-state semantics

When a branch of a nondeterministic computation runs into a dead end and the continuation is picked up at the most recent branching point, any alterations made to the state by the terminated branch are invisible to the continuation. We refer to this semantics as *local-state semantics*. Gibbons and Hinze (2011) also call it backtrackable state.

The local-state laws. The following two laws characterize the local-state semantics for a monad with state and nondeterminism:

$$\textbf{put-right-identity}: \quad put\ s \gg \varnothing = \varnothing, \tag{4.1}$$

$$\textbf{put-left-distributivity}: \quad put\ s \gg (m_1 \mathbin{[]} m_2) = (put\ s \gg m_1) \mathbin{[]} (put\ s \gg m_2). \tag{4.2}$$

The equation (4.1) expresses that $\varnothing$ is the right identity of *put*; it annihilates state updates. The other law expresses that *put* distributes from the left in ($[]$).

These two laws only focus on the interaction between *put* and nondeterminism. The following laws for *get* can be derived from other laws. The proof can be found in Appendix 1.

$$\textbf{get-right-identity}: \quad get \gg \varnothing = \varnothing, \tag{4.3}$$

$$\textbf{get-left-distributivity}: \quad get \gg\!= (\lambda x \to k_1\ x \mathbin{[]} k_2\ x) = (get \gg\!= k_1) \mathbin{[]} (get \gg\!= k_2). \tag{4.4}$$

If we take these four equations together with the left-identity and right-distributivity laws of nondeterminism, we can say that nondeterminism and state "commute"; if a *get* or *put* precedes a $\varnothing$ or $[]$, we can exchange their order (and vice versa).

Implementation. Implementation-wise, the laws imply that each nondeterministic branch has its own copy of the state. For instance, Equation (4.2) gives us

$$put\ 42\ (put\ 21 \mathbin{[]} get) = (put\ 42 \gg put\ 21) \mathbin{[]} (put\ 42 \gg get)$$

The state we *get* in the second branch is still 42, despite the *put* 21 in the first branch.

One implementation satisfying the laws is

$$\textbf{type}\ Local\ s\ m\ a = s \to m\ (a, s)$$

where m is a nondeterministic monad, the simplest structure of which is a list. This implementation is exactly that of *StateT s m a* in the Monad Transformer Library (Jones, 1995) which we have introduced in Section 3.2.

With effect handling (Kiselyov and Ishii, 2015; Wu et al., 2014), we get the local state semantics when we run the state handler before the nondeterminism handler:

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$h_{Local} :: Functor\, f \Rightarrow Free\,(State_F\, s :+: Nondet_F :+: f)\, a \to (s \to Free\, f\, [a])$$
$$h_{Local} = fmap\,(fmap\,(fmap\, fst) \circ h_{ND+f}) \circ run_{StateT} \circ h_{State}$$

In the case where the remaining signature is empty ($f = Nil_F$), we get:

$$fmap\, h_{Nil} \circ h_{Local} :: Free\,(State_F\, s :+: Nondet_F :+: Nil_F)\, a \to (s \to [a])$$

Here, the result type $(s \to [a])$ differs from $s \to [(a, s)]$ in that it produces only a list of results ($[a]$) and not pairs of results and their final state ($[(a, s)]$). The latter is needed for *Local s m* to have the structure of a monad, in particular to support the modular composition of computations with $(\gg\!=)$. Such is not needed for the carriers of handlers because the composition of computations is taken care of by the $(\gg\!=)$ operator of the free monad.

4.2 Global-state semantics

Alternatively, one 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 branches of nondeterministic computation, we call this state the *global-state semantics*.

The global-state law. The global-state semantics sets apart non-backtrackable state from backtrackable state. In addition to the general laws for nondeterminism ((2.6) – (2.9)) and state ((2.10) – (2.13)), we provide a *global-state law* to govern the interaction between nondeterminism and state.

$$\textbf{put-or}: \quad (put\, s \gg m) \,[\!]\, n = put\, s \gg (m \,[\!]\, n). \tag{4.5}$$

This law allows lifting a *put* operation from the left branch of a nondeterministic choice. For instance, if $m = \varnothing$ in the left-hand side of the equation, then under local-state semantics (laws (2.6) and (4.1)) the left-hand side becomes equal to n, whereas under global-state semantics (laws (2.6) and (4.5)) the equation simplifies to $put\, s \gg n$.

Implementation. Figuring out a correct implementation for the global-state monad is tricky. One might believe that $Global\, s\, m\, a = s \to (m\, a, s)$ is a natural implementation of such a monad. However, the usual naive implementation of $(\gg\!=)$ for it does not satisfy right-distributivity (2.8) and is therefore not even a monad. The type *ListT* (*State s*) from the Monad Transformer Library (Jones, 1995) expands to essentially the same implementation with monad *m* instantiated by the list monad. This implementation has the same flaws. More careful implementations of *ListT* (often referred to as "*ListT* done right") satisfying right-distributivity (2.8) and other monad laws have been proposed by Volkov (2014); Gale (2007). The following implementation is essentially that of Gale.

newtype $Global\, s\, a = Gl\, \{runGl :: s \to (Maybe\,(a, Global\, s\, a), s)\}$

The *Maybe* in this type indicates that a computation may fail to produce a result. However, since the *s* is outside of the *Maybe*, a modified state is returned even if the computation fails. This *Global s a* type is an instance of the *MState* and *MNondet* type classes.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

```
instance MNondet (Global s) where
  ∅     = Gl (λs → (Nothing, s))
  p ▯ q = Gl (λs → case runGl p s of
    (Nothing,   t) → runGl q t
    (Just (x, r), t) → (Just (x, r ▯ q), t))

instance MState s (Global s) where
  get   = Gl (λs → (Just (s, ∅), s))
  put s = Gl (λ_ → (Just ((), ∅), s))
```

Failure, of course, returns an empty continuation and an unmodified state. Branching first exhausts the left branch before switching to the right branch.

Effect handlers (Kiselyov and Ishii, 2015; Wu et al., 2014) also provide implementations that match our intuition of non-backtrackable computations. The global-state semantics can be implemented by simply switching the order of the two effect handlers compared to the local state handler h_{Local}.

$$h_{Global} :: (Functor\ f) \Rightarrow Free\ (State_F\ s :+: Nondet_F :+: f)\ a \to (s \to Free\ f\ [a])$$
$$h_{Global} = fmap\ (fmap\ fst) \circ run_{StateT} \circ h_{State} \circ h_{ND+f} \circ (\Leftrightarrow)$$

This also runs a single state through a nondeterministic computation. Here, the $(\Leftrightarrow)$ isomorphism from Section 3.2 allows h_{Local} and h_{Global} have the same type signature.

In the case where the remaining signature is empty ($f = Nil_F$), we get:

$$fmap\ h_{Nil} \circ h_{Global} :: Free\ (State_F\ s :+: Nondet_F :+: Nil_F)\ a \to (s \to [a])$$

Like in Section 4.1, the carrier type here is again simpler than that of the corresponding monad because it does not have to support the $(\gg\!=)$ operator.

4.3 Simulating local state with global state

Both local state and global state have their own laws and semantics. Also, both interpretations of nondeterminism with state have their own advantages and disadvantages.

Local-state semantics imply that each nondeterministic branch has its own state. This may be expensive if the state is represented by data structures, e.g. arrays, that are costly to duplicate. For example, when each new state is only slightly different from the previous, we have a wasteful duplication of information.

Global-state semantics, however, threads a single state through the entire computation without making any implicit copies. Consequently, it is easier to control resource usage and apply optimization strategies in this setting. However, doing this to a program that has a backtracking structure, and would be more naturally expressed in a local-state style, comes at a great loss of clarity. Furthermore, it is significantly more challenging for programmers to reason about global-state semantics than local-state semantics.

To resolve this dilemma, we can write our programs in a local-state style and then translate them to the global-state style to enable low-level optimizations. In this subsection, we show one systematic program translation that alters a program written for local-state semantics to a program that, when interpreted under global-state semantics, behaves exactly the same as the original program interpreted under local-state semantics. This translation explicitly copies the whole state and relies on the nondeterminism mechanism to insert state-restoring branches. We will show other translations from local-state semantics to global-state semantics which avoid the copying and do not rely on nondeterminism in Sections 7 and 8.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

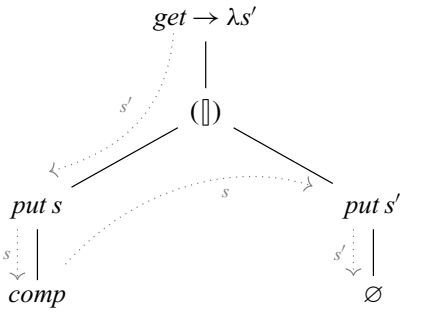


Fig. 1: State-restoring put-operation.

State-restoring put. Central to the implementation of backtracking in the global state setting is the backtracking variant put_R of put. The idea is that put_R, when run with a global state, satisfies laws (2.10) to (4.2) — the state laws and the local-state laws. Going forward, put_R modifies the state as usual, but, when backtracked over, it restores the old state.

We implement put_R using both state and nondeterminism as follows:

$$put_R :: (MState\ s\ m, MNondet\ m) \Rightarrow s \to m\ ()$$
$$put_R\ s = get \gg\!= \lambda s' \to put\ s \mathbin{[\!]} side\ (put\ s')$$

Here the *side* branch is executed for its side effect only; it fails before yielding a result.

$$side :: MNondet\ m \Rightarrow m\ a \to m\ b$$
$$side\ m = m \gg \varnothing$$

Intuitively, the second branch generated by put_R can be understood as a backtracking or state-restoring branch. The $put_R\ s$ operation changes the state to s in the first branch $put\ s$, and then restores it to the original state s' in the second branch after we finish all computations in the first branch. Then, the second branch immediately fails so that we can keep going to other branches with the original state s'. For example, assuming an arbitrary computation *comp* is placed after a state-restoring put, we have the following calculation.

$$\begin{aligned}
&put_R\ s \gg comp \\
=\ &\{\text{- definition of } put_R \text{ -}\} \\
&(get \gg\!= \lambda s' \to put\ s \mathbin{[\!]} side\ (put\ s')) \gg comp \\
=\ &\{\text{- right-distributivity (2.8) -}\} \\
&(get \gg\!= \lambda s' \to (put\ s \gg comp) \mathbin{[\!]} (side\ (put\ s') \gg comp)) \\
=\ &\{\text{- left identity (2.9) -}\} \\
&(get \gg\!= \lambda s' \to (put\ s \gg comp) \mathbin{[\!]} side\ (put\ s'))
\end{aligned}$$

This program saves the current state s', computes *comp* using state s, and then restores the saved state s'. Figure 1 shows how the state-passing works and the flow of execution for a computation after a state-restoring put.

Another example of put_R is shown in Table 2, where three programs are run with initial state s_0. Note the difference between the final state and the program result for the state-restoring put.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Table 2: Comparing *put* and put_R

	Program result	Final state
$\eta\, x \gg get$	s_0	s_0
$put\, s \gg \eta\, x \gg get$	s	s
$put_R\, s \gg \eta\, x \gg get$	s	s_0

Translation with state-restoring put.

We do not expect the programmer to program against the global-state semantics directly and use the state-restoring put_R as they see fit, as this can be quite confusing and error-prone. Instead, we provide an automatic translation: The programmer writes their program against the local-state semantics and uses the regular *put*. We then translate the local-state semantics program to a corresponding global-state semantics program using the effect handler *local2global*:

$$
\begin{aligned}
&local2global :: Functor\, f \\
&\qquad \Rightarrow Free\,(State_F\, s :+: Nondet_F :+: f)\, a \\
&\qquad \rightarrow Free\,(State_F\, s :+: Nondet_F :+: f)\, a \\
&local2global = fold\, Var\, alg \\
&\quad \textbf{where} \\
&\qquad alg\,(Inl\,(Put\, t\, k)) = put_R\, t \gg k \\
&\qquad alg\, p = Op\, p
\end{aligned}
$$

This handler maps the *put* with local-state semantics onto the state-restoring put_R with global-state semantics. All other local-state operations are mapped onto their global-state counterpart.

For example, recall the backtracking algorithm *queens* for the n-queens example in Section 2.3. It is initially designed to run in the local-state semantics because every branch maintains its own copy of the state and has no influence on other branches. We can handle it with h_{Local} as follows.

$$
\begin{aligned}
&queens_{Local} :: Int \rightarrow [[Int]] \\
&queens_{Local} = h_{Nil} \circ flip\, h_{Local}\, (0, [\,]) \circ queens
\end{aligned}
$$

With the simulation *local2global*, we can also translate *queens* to an equivalent program in global-state semantics and handle it with h_{Global}.

$$
\begin{aligned}
&queens_{Global} :: Int \rightarrow [[Int]] \\
&queens_{Global} = h_{Nil} \circ flip\, h_{Global}\, (0, [\,]) \circ local2global \circ queens
\end{aligned}
$$

The following theorem guarantees that the translation *local2global* preserves the meaning when switching from local-state to global-state semantics:

Theorem 1. $h_{Global} \circ local2global = h_{Local}$

Proof Both the left-hand side and the right-hand side of the equation consist of function compositions involving one or more folds. We apply fold fusion separately on both sides to contract each into a single fold:

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$\begin{aligned} h_{Global} \circ local2global &= fold\ gen_{LHS}\ (alg^{S}_{LHS} \# alg^{ND}_{RHS} \# fwd_{LHS}) \\ h_{Local} &= fold\ gen_{RHS}\ (alg^{S}_{RHS} \# alg^{ND}_{RHS} \# fwd_{RHS}) \end{aligned}$$

We approach this calculationally. That is to say, we do not first postulate definitions of the unknowns above (alg^{S}_{LHS} and so on) and then verify whether the fusion conditions are satisfied. Instead, we discover the definitions of the unknowns. We start from the known side of each fusion condition and perform case analysis on the possible shapes of input. By simplifying the resulting case-specific expression, and pushing the handler applications inwards, we end up at a point where we can read off the definition of the unknown that makes the fusion condition hold for that case.

Finally, we show that both folds are equal by showing that their corresponding parameters are equal:

$$\begin{aligned} gen_{LHS} &= gen_{RHS} \\ alg^{S}_{LHS} &= alg^{S}_{RHS} \\ alg^{ND}_{LHS} &= alg^{ND}_{RHS} \\ fwd_{LHS} &= fwd_{RHS} \end{aligned}$$

A noteworthy observation is that, for fusing the left-hand side of the equation, we do not use the standard fusion rule **fusion-post** (3.2):

$$\begin{aligned} h_{Global} \circ fold\ Var\ alg &= fold\ (h_{Global} \circ Var)\ alg' \\ \Leftarrow \qquad h_{Global} \circ alg &= alg' \circ fmap\ h_{Global} \end{aligned}$$

where $local2global = fold\ Var\ alg$. The problem is that we will not find an appropriate alg' such that $alg'\ (fmap\ h_{Global}\ t)$ restores the state for any t of type $(State_F\ s :+: Nondet_F :+: f)\ (Free\ (State_F\ s :+: Nondet_F :+: f)\ a)$.

Fortunately, we do not need such an alg'. We can assume that the subterms of t have already been transformed by *local2global*, and thus all occurrences of *Put* appear in the put_R constellation.

We can incorporate this assumption by using the alternative fusion rule **fusion-post'** (3.3):

$$\begin{aligned} h_{Global} \circ fold\ Var\ alg &= fold\ (h_{Global} \circ Var)\ alg' \\ \Leftarrow \qquad h_{Global} \circ alg \circ fmap\ local2global &= alg' \circ fmap\ h_{Global} \circ fmap\ local2global \end{aligned}$$

The additional $fmap\ local2global$ in the condition captures the property that all the subterms have been transformed by *local2global*.

In order to not clutter the proofs, we abstract everywhere over this additional $fmap\ local2global$ application, except for the key lemma which expresses that the syntactic transformation *local2global* makes sure that, despite any temporary changes, the computation t restores the state back to its initial value.

We elaborate each of these steps in Appendix 2. ■

Note on global replacement. To preserve the behaviour when going from local-state to global-state semantics, care should be taken to replace *all* occurrences of *put*. Particularly, placing a program in a larger context, where *put* has not been replaced, can change the

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

meaning of its subprograms. An example of such a problematic context is $(\gg put\ t)$, where the *get-put* law (2.12) breaks and programs $get \gg put_R$ and $\eta\,()$ can be differentiated:

$$
\begin{array}{ll}
 & (get \gg put_R) \gg put\ t \\
= & \quad \{\text{- definition of } put_R \text{ -}\} \\
 & (get \gg\!= \lambda s \to get \gg\!= \lambda s_0 \to put\ s \,[\!]\, side\ (put\ s_0)) \gg put\ t \\
= & \quad \{\text{- get-get (2.13) -}\} \\
 & (get \gg\!= \lambda s \to put\ s \,[\!]\, side\ (put\ s)) \gg put\ t \\
= & \quad \{\text{- right-distributivity (2.8) -}\} \\
 & (get \gg\!= \lambda s \to (put\ s \gg put\ t) \,[\!]\, (side\ (put\ s)) \gg put\ t) \\
= & \quad \{\text{- left-identity (2.9) -}\} \\
 & (get \gg\!= \lambda s \to (put\ s \gg put\ t) \,[\!]\, side\ (put\ s)) \\
= & \quad \{\text{- put-put (2.10) -}\} \\
 & (get \gg\!= \lambda s \to put\ t \,[\!]\, side\ (put\ s))
\end{array}
\qquad
\begin{array}{l}
\eta\,() \gg put\ t \\
= put\ t
\end{array}
$$

Those two programs do not behave in the same way when $s \not\equiv t$. Hence, only provided that *all* occurrences of *put* in a program are replaced by put_R can we simulate local-state semantics with global-state semantics. This has been articulated in the proof by the composition $h_{Global} \circ local2global$: there is no room between the replacement by *local2global* and the interpretation with h_{Global} to add plain *put* operations. The global replacement requirement also manifests itself in the proof, in the form of the **fusion-post'** rule rather than the more widely used **fusion-post** rule.

5 Modelling nondeterminism with state

In the previous section, we have translated the local-state semantics, a high-level combination of the state and nondeterminism effects, to the global-state semantics, a low-level combination of the state and nondeterminism effects. In this section, we further translate the resulting nondeterminism component, which is itself a relatively high-level effect, to a lower-level implementation with the state effect. Our translation coincides with the fact that, while nondeterminism is typically modelled using the *List* monad, many efficient nondeterministic systems, such as Prolog, use a low-level state-based implementation to implement the nondeterminism mechanism.

5.1 Simulating nondeterminism with state

The main idea of simulating nondeterminism with state is to explicitly manage

1. a list of the *results* found so far, and
2. a list of yet to be explored branches, which we call a *stack*.

This stack corresponds to the choicepoint stack in Prolog. When entering one branch, we can push other branches to the stack. When leaving the branch, we collect its result and pop a new branch from the stack to continue.

We define a new type $S\ a$ consisting of the *results* and *stack*.

type $Comp\ s\ a = Free\ (State_F\ s)\ a$
data $S\ a = S\ \{results :: [a], stack :: [Comp\ (S\ a)\ ()]\}$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

```
pop_S :: Comp (S a) ()
pop_S = do
  S xs stack ← get
  case stack of
    []     → η ()
    p : ps → do
      put (S xs ps); p
```

(a) Popping from the stack.

```
push_S :: Comp (S a) ()
       → Comp (S a) ()
       → Comp (S a) ()
push_S q p = do
  S xs stack ← get
  put (S xs (q : stack))
  p
```

(b) Pushing to the stack.

```
append_S :: a
         → Comp (S a) ()
         → Comp (S a) ()
append_S x p = do
  S xs stack ← get
  put (S (xs ++[x]) stack)
  p
```

(c) Appending a result.

Fig. 2: Auxiliary functions pop_S, $push_S$, and $append_S$.

The branches in the stack are represented by computations in the form of free monads over the $State_F$ signature. We do not allow branches to use other effects here to show the idea more clearly. In Section 5.2, we will consider the more general case where branches can have any effects abstracted by a functor f.

For brevity, instead of defining a new stack effect capturing the stack operations like pop and push, we implement stack operations with the state effect. We define three auxiliary functions in Figure 2 to interact with the stack in $S\,a$:

- The function pop_S removes and executes the top element of the stack.
- The function $push_S$ pushes a branch into the stack.
- The function $append_S$ adds a result to the current results.

Now, everything is in place to define a simulation function $nondet2state_S$ that interprets nondeterministic programs represented by the free monad $Free\ Nondet_F\ a$ as state-wrapped programs represented by the free monad $Free\ (State_F\ (S\ a))\ ()$.

```
nondet2state_S :: Free Nondet_F a → Free (State_F (S a)) ()
nondet2state_S = fold gen alg
  where
    gen x       = append_S x pop_S
    alg Fail    = pop_S
    alg (Or p q) = push_S q p
```

The generator of this handler records a new result and then pops the next branch from the stack and proceeds with it. Likewise, for failure the handler simply pops and proceeds with the next branch. For nondeterministic choices, the handler pushes the second branch on the stack and proceeds with the first branch. The $nondet2state_S$ implements the depth-first search strategy which is consistent with the implementation of h_{ND}[3].

To extract the final result from the S wrapper, we define the $extract_S$ function.

```
extract_S :: State (S a) () → [a]
extract_S x = results ∘ snd $ run_State x (S [] [])
```

[3] It is also possible to implement the breadth-first search strategy by replacing the stack with a queue.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Finally, we define the function run_{ND} which wraps everything up to handle a nondeterministic computation to a list of results. The state handler h'_{State} is defined in Section 3.1.

$$run_{ND} :: Free\ Nondet_F\ a \to [a]$$
$$run_{ND} = extract_S \circ h'_{State} \circ nondet2state_S$$

We have the following theorem showing the correctness of the simulation via the equivalence of the run_{ND} function and the nondeterminism handler h_{ND} defined in Section 3.1.

Theorem 2. $run_{ND} = h_{ND}$

The proof can be found in Appendix 3.1. The main idea is again to use fold fusion. Consider the expanded form

$$(extract_S \circ h'_{State}) \circ nondet2state_S = h_{ND}$$

Both $nondet2state_S$ and h_{ND} are written as folds. We use the law **fusion-post'** (3.3) to fuse the left-hand side into a single fold. Since the right-hand side is already a fold, to prove the equation we just need to check the components of the fold h_{ND} satisfy the conditions of the fold fusion, i.e., the following two equations: For the latter, we only need to prove the following two equations:

$$\begin{aligned} & (extract_S \circ h'_{State}) \circ gen = gen_{ND} \\ & (extract_S \circ h'_{State}) \circ alg \circ fmap\ nondet2state_S \\ = \quad & alg_{ND} \circ fmap\ (extract_S \circ h'_{State}) \circ fmap\ nondet2state_S \end{aligned}$$

where gen and alg are from the definition of $nondet2state_S$, and gen_{ND} and alg_{ND} are from the definition of h_{ND}.

5.2 Combining the simulation with other effects

The $nondet2state_S$ function only considers nondeterminism as the only effect. In this section, we generalize it to work in combination with other effects. One immediate benefit is that we can use it in together with our previous simulation *local2global* in Section 4.3.

Firstly, we need to augment the signature in the computation type for branches with an additional functor f for other effects. The computation type is essentially changed from $Free\ (State_F\ s)\ a$ to $Free\ (State_F\ s :+: f)\ a$. We define the state type $SS\ f\ a$ as follows:

$$\textbf{type}\ Comp_{SS}\ s\ f\ a = Free\ (State_F\ s :+: f)\ a$$
$$\textbf{data}\ SS\ f\ a = SS\ \{results_{SS} :: [a], stack_{SS} :: [Comp_{SS}\ (SS\ f\ a)\ f\ ()]\}$$

We also modify the three auxiliary functions in Figure 2 to pop_{SS}, $push_{SS}$ and $append_{SS}$ in Figure 3. They are almost the same as the previous versions apart from being adapted to use the new state-wrapper type $SS\ f\ a$.

The simulation function *nondet2state* is also very similar to $nondet2state_S$ except for requiring a forwarding algebra *fwd* to deal with the additional effects in f.

$$nondet2state :: Functor\ f \Rightarrow Free\ (Nondet_F :+: f)\ a \to Free\ (State_F\ (SS\ f\ a) :+: f)\ ()$$
$$nondet2state = fold\ gen\ (alg \,\#\, fwd)$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

```
popSS :: Functor f
  ⇒ CompSS (SS f a) f ()
popSS = do
  SS xs stack ← get
  case stack of
    []     → η ()
    p : ps → do
      put (SS xs ps); p
```

(a) Popping from the stack.

```
pushSS :: Functor f
  ⇒ CompSS (SS f a) f ()
  → CompSS (SS f a) f ()
  → CompSS (SS f a) f ()
pushSS q p = do
  SS xs stack ← get
  put (SS xs (q : stack))
  p
```

(b) Pushing to the stack.

```
appendSS :: Functor f
  ⇒ a
  → CompSS (SS f a) f ()
  → CompSS (SS f a) f ()
appendSS x p = do
  SS xs stack ← get
  put (SS (xs ++[x]) stack)
  p
```

(c) Appending a result.

Fig. 3: Auxiliary functions pop_{SS}, $push_{SS}$ and $append_{SS}$.

```
  where
    gen x       = appendSS x popSS
    alg Fail    = popSS
    alg (Or p q) = pushSS q p
    fwd y       = Op (Inr y)
```

The function run_{ND+f} puts everything together: it translates the nondeterminism effect into the state effect and forwards other effects using *nondet2state*, then handles the state effect using h_{State}, and finally extracts the results from the final state using $extract_{SS}$.

```
runND+f :: Functor f ⇒ Free (Nondet_F :+: f) a → Free f [a]
runND+f = extractSS ∘ hState ∘ nondet2state
extractSS :: Functor f ⇒ StateT (SS f a) (Free f) () → Free f [a]
extractSS x = resultsSS ∘ snd ⟨$⟩ runStateT x (SS [] [])
```

We have the following theorem showing that the simulation run_{ND+f} is equivalent to the modular nondeterminism handler h_{ND+f} in Section 3.2.

Theorem 3. $run_{ND+f} = h_{ND+f}$

The proof proceeds essentially in the same way as in the non-modular setting. The main difference, due to the modularity, is an additional proof case for the forwarding algebra.

$$\begin{aligned} & (extract_{SS} \circ h_{State}) \circ fwd \circ fmap\ nondet2state_S \\ = \quad & fwd_{ND+f} \circ fmap\ (extract_{SS} \circ h_{State}) \circ fmap\ nondet2state_S \end{aligned}$$

The full proof can be found in Appendix 3.2.

6 All in one

This section combines the results of the previous two sections to ultimately simulate the combination of nondeterminism and state with a single state effect.

6.1 Modelling two states with one state

When we combine the two simulation steps from the two previous sections, we end up with a computation that features two state effects. The first state effect is the one present

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

originally, and the second state effect keeps track of the results and the stack of remaining branches to simulate the nondeterminism.

For a computation of type $Free\,(State_F\,s_1 :+: State_F\,s_2 :+: f)\,a$ that features two state effects, we can go to a slightly more primitive representation $Free\,(State_F\,(s_1, s_2) :+: f)\,a$ featuring only a single state effect that is a pair of the previous two states.

The handler *states2state* implements the simulation by projecting different *get* and *put* operations to different components of the pair of states.

$$
\begin{aligned}
&states2state :: Functor\,f \\
&\qquad \Rightarrow Free\,(State_F\,s_1 :+: State_F\,s_2 :+: f)\,a \\
&\qquad \rightarrow Free\,(State_F\,(s_1, s_2) :+: f)\,a \\
&states2state = fold\;Var\;(alg_1 \,\#\, alg_2 \,\#\, fwd) \\
&\quad \textbf{where} \\
&\qquad alg_1\,(Get\,k) \;\; = get \gg\!= \lambda(s_1, \_) \rightarrow k\,s_1 \\
&\qquad alg_1\,(Put\,s_1'\,k) = get \gg\!= \lambda(\_, s_2) \rightarrow put\,(s_1', s_2) \gg k \\
&\qquad alg_2\,(Get\,k) \;\; = get \gg\!= \lambda(\_, s_2) \rightarrow k\,s_2 \\
&\qquad alg_2\,(Put\,s_2'\,k) = get \gg\!= \lambda(s_1, \_) \rightarrow put\,(s_1, s_2') \gg k \\
&\qquad fwd\;op \qquad\quad = Op\,(Inr\;op)
\end{aligned}
$$

We have the following theorem showing the correctness of *states2state*:

Theorem 4. $h_{States} = nest \circ h_{State} \circ states2state$

On the left-hand side, we write h_{States} for the composition of two consecutive state handlers:

$$
\begin{aligned}
&h_{States} :: Functor\,f \Rightarrow Free\,(State_F\,s_1 :+: State_F\,s_2 :+: f)\,a \rightarrow StateT\,s_1\,(StateT\,s_2\,(Free\,f))\,a \\
&h_{States}\,x = StateT\,(h_{State} \circ run_{StateT}\,(h_{State}\,x)) \\
&h'_{States} :: Functor\,f \Rightarrow Free\,(State_F\,s_1 :+: State_F\,s_2 :+: f)\,a \rightarrow StateT\,(s_1, s_2)\,(Free\,f)\,a \\
&h'_{States}\,t = StateT\;\$\;\lambda(s_1, s_2) \rightarrow \alpha\;\langle\$\rangle\;run_{StateT}\,(h_{State}\,(run_{StateT}\,(h_{State}\,t)\,s_1))\,s_2
\end{aligned}
$$

On the right-hand side, we use the isomorphism *nest* to mediate between the two different carrier types. The definition of *nest* and its inverse *flatten* are defined as follows:

$$
\begin{aligned}
&nest \;\; :: Functor\,f \Rightarrow StateT\,(s_1, s_2)\,(Free\,f)\,a \rightarrow StateT\,s_1\,(StateT\,s_2\,(Free\,f))\,a \\
&nest\,t \;\; = StateT\;\$\;\lambda s_1 \rightarrow StateT\;\$\;\lambda s_2 \rightarrow \alpha^{-1}\;\langle\$\rangle\;run_{StateT}\,t\,(s_1, s_2) \\
&flatten \;\; :: Functor\,f \Rightarrow StateT\,s_1\,(StateT\,s_2\,(Free\,f))\,a \rightarrow StateT\,(s_1, s_2)\,(Free\,f)\,a \\
&flatten\,t = StateT\;\$\;\lambda(s_1, s_2) \rightarrow \alpha\;\langle\$\rangle\;run_{StateT}\,(run_{StateT}\,t\,s_1)\,s_2
\end{aligned}
$$

where the isomorphism α^{-1} and its inverse α rearrange a nested tuple

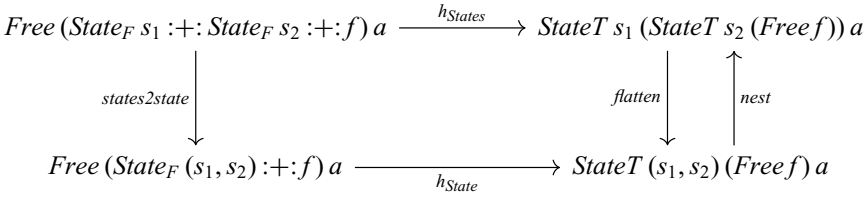
$$
\begin{aligned}
&\alpha :: ((a, x), y) \rightarrow (a, (x, y)) \\
&\alpha\,((a, x), y) = (a, (x, y))
\end{aligned}
\qquad\qquad
\begin{aligned}
&\alpha^{-1} :: (a, (x, y)) \rightarrow ((a, x), y) \\
&\alpha^{-1}\,(a, (x, y)) = ((a, x), y)
\end{aligned}
$$

The proof of Theorem 4 can be found in Appendix 4. Theorem 4 has two function compositions on the right-hand side, which would require using fusion twice, resulting in a complicated handler. To avoid this complexity, we show the correctness of the isomorphism of *nest* and *flatten*, and prove the following equation:

$$flatten \circ h_{States} = h_{State} \circ states2state$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

The following commuting diagram summarizes the simulation.



6.2 Putting everything together

We have defined three translations for encoding high-level effects as low-level effects.

- The function *local2global* simulates the high-level local-state semantics with global-state semantics for the nondeterminism and state effects (Section 4).
- The function *nondet2state* simulates the high-level nondeterminism effect with the state effect (Section 5).
- The function *states2state* simulates multiple state effects with a single state effect (Section 6.1).

Combining them, we can encode the local-state semantics for nondeterminism and state with just one state effect. The ultimate simulation function *simulate* is defined as follows:

$$
\begin{aligned}
&\mathit{simulate} :: \mathit{Functor}\ f \\
&\qquad \Rightarrow \mathit{Free}\ (\mathit{State}_F\ s :+: \mathit{Nondet}_F :+: f)\ a \\
&\qquad \rightarrow s \rightarrow \mathit{Free}\ f\ [a] \\
&\mathit{simulate} = \mathit{extract} \circ h_{\mathit{State}} \circ \mathit{states2state} \circ \mathit{nondet2state} \circ (\Leftrightarrow) \circ \mathit{local2global}
\end{aligned}
$$

Similar to the $\mathit{extract}_{SS}$ function in Section 5.2, we use the *extract* function to get the final results from the final state.

$$
\begin{aligned}
&\mathit{extract} :: \mathit{Functor}\ f \\
&\qquad \Rightarrow \mathit{StateT}\ (\mathit{SS}\ (\mathit{State}_F\ s :+: f)\ a, s)\ (\mathit{Free}\ f)\ () \\
&\qquad \rightarrow s \rightarrow \mathit{Free}\ f\ [a] \\
&\mathit{extract}\ x\ s = \mathit{results}_{SS} \circ \mathit{fst} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ x\ (\mathit{SS}\ [\,]\ [\,], s)
\end{aligned}
$$

Figure 4 illustrates each step of this simulation.

In the *simulate* function, we first use our three simulations *local2global*, *nondet2state* and *states2state* to interpret the local-state semantics for state and nondeterminism in terms of only one state effect. Then, we use the handler h_{State} to interpret the state effect into a state monad transformer. Finally, we use the function *extract* to get the final results.

We have the following theorem showing that the *simulate* function exactly behaves the same as the local-state semantics given by h_{Local}.

Theorem 5. $\mathit{simulate} = h_{Local}$

The proof can be found in Appendix 5.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{array}{c}
\mathit{Free}\,(\mathit{State}_F\ s :+: \mathit{Nondet}_F :+: f)\ a \\
\downarrow{\scriptstyle \mathit{local2global}} \\
\mathit{Free}\,(\mathit{State}_F\ s :+: \mathit{Nondet}_F :+: f)\ a \\
\downarrow{\scriptstyle (\Leftrightarrow)} \\
\mathit{Free}\,(\mathit{Nondet}_F :+: \mathit{State}_F\ s :+: f)\ a \\
\downarrow{\scriptstyle \mathit{nondet2state}} \\
\mathit{Free}\,(\mathit{State}_F\ (\mathit{SS}\ (\mathit{State}_F\ s :+: f)\ a) :+: \mathit{State}_F\ s :+: f)\ () \\
\downarrow{\scriptstyle \mathit{states2state}} \\
\mathit{Free}\,(\mathit{State}_F\ (\mathit{SS}\ (\mathit{State}_F\ s :+: f)\ a, s) :+: f)\ () \\
\downarrow{\scriptstyle h_{\mathit{State}}} \\
\mathit{StateT}\ (\mathit{SS}\ (\mathit{State}_F\ s :+: f)\ a, s)\ (\mathit{Free}\ f)\ () \\
\downarrow{\scriptstyle \mathit{extract}} \\
s \to \mathit{Free}\ f\ [a]
\end{array}
$$

Fig. 4: An overview of the *simulate* function.

We provide a more compact and direct definition of *simulate* by fusing all the consecutive steps into a single handler:

```
type Comp s f a = (CP f a s, s) → Free f [a]
data CP f a s = CP {results :: [a], cpStack :: [Comp s f a]}

simulate_F :: Functor f
           ⇒ Free (State_F s :+: Nondet_F :+: f) a
           → s
           → Free f [a]
simulate_F x s = fold gen (alg_1 # alg_2 # fwd) x (CP [] [], s)
  where
    gen x          (CP xs stack, s) = continue (xs ++ [x]) stack s
    alg_1 (Get k)  (CP xs stack, s) = k s (CP xs stack, s)
    alg_1 (Put t k)(CP xs stack, s) = k (CP xs (backtracking s : stack), t)
    alg_2 Fail     (CP xs stack, s) = continue xs stack s
    alg_2 (Or p q) (CP xs stack, s) = p (CP xs (q : stack), s)
    fwd op         (CP xs stack, s) = Op (fmap ($(CP xs stack, s)) op)
    backtracking s (CP xs stack, _) = continue xs stack s
    continue xs stack s             = case stack of
                                        []     → η xs
                                        (p:ps) → p (CP xs ps, s)
```

The common carrier of the above algebras $alg_1 \# alg_2 \# fwd$ is $Comp\ s\ f\ a$. This is a computation that takes the current results, choicepoint stack and application state, and returns the list of all results. The first two inputs are bundled in the *CP* type.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

N-queens with only one state. With *simulate*, we can implement the backtracking algorithm of the n-queens problem in Section 2.3 with only one state effect as follows.

$$queens_{Sim} :: Int \to [[Int]]$$
$$queens_{Sim} = h_{Nil} \circ flip\ simulate\ (0, [\,]) \circ queens$$

7 Modelling local state with undo

Section 4.3 uses *local2global* to simulate local state with global state by replacing *put* with the state-restoring version put_R. The put_R operation makes the implicit state copying of the local-state semantics explicit in the global-state semantics. This copying can be rather costly if the state is big (e.g., a long array). It is especially wasteful when the modifications made to that state are small (e.g., a single entry in the array). Fortunately, lower-level effects present more opportunities for fine-grained optimization. In particular, we can exploit the global-state semantics to avoid copying the whole state. Instead, we only keep track of the modifications made to the state, and undo them when backtracking. This section formalizes that approach in terms of an alternative translation from the local-state semantics to the global-state semantics that incrementally records *reversible state updates*.

7.1 Reversible state updates

Our goal is to undo a state change without holding on to the old state. Instead, we should be able to recover the old state from the new state. However, knowing only the new state is usually not enough to accomplish this. We must also know "what update was applied to the old state that led to the new state".

We reify the information about the update in a type u, which depends on the particular application at hand. For example, in the *queens* program of Section 2.3 we repeatedly update the state to place an additional queen on the board. Recall that a state s of type $(Int, [Int])$ consists of the current column c and the partial solution *sol*, i.e., the rows of the already placed queens. Hence, the information we need to characterize an update is the row r of the queen to place in the current column, i.e., $u = Int$. The update itself is performed as $s \oplus r$, where

$$(\oplus) :: (Int, [Int]) \to Int \to (Int, [Int])$$
$$(\oplus)\ (c, sol)\ r = (c + 1, r : sol)$$

Now we can clearly recover the old state from the new state and the modification as follows:

$$(\ominus) :: (Int, [Int]) \to Int \to (Int, [Int])$$
$$(\ominus)\ (c, sol)\ r = (c - 1, tail\ sol)$$

Indeed, we clearly have $(s \oplus r) \ominus r = s$.

In general, we define a typeclass *Undo s u* with two operations $(\oplus)$ and $(\ominus)$ to characterize reversible state updates. Here, s is the type of states and u is the type of updates.

class *Undo s u* **where**
$$(\oplus) :: s \to u \to s$$
$$(\ominus) :: s \to u \to s$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Instances of *Undo* should satisfy the following law which says that $\ominus x$ is a left inverse of $\oplus x$:

$$\textbf{plus-minus}: \quad (\ominus x) \circ (\oplus x) \;=\; id. \tag{7.1}$$

7.2 Reversible state update effect

For our optimized approach to work, we have to restrict the way the state is changed in local-state programs. We no longer allow arbitrary *put* s' calls to change the implicit state. The only supported changes are of the form *put* $(s \oplus u)$ where s is the current state.

To enforce this requirement, we replace the general *get*/*put* interface provided by *MState* with the more restricted interface of a new type class:

```
class (Monad m, Undo s u) ⇒ MModify s u m | m → s, m → u where
  mget    :: m s
  update  :: u → m ()
  restore :: u → m ()
```

This *MModify* class has three operations: a *mget* operation that reads and returns the state (similar to the *get* operation of *MState*), an *update u* operation that updates the state with the reversible state change u, and a *restore u* operations that reverses the update u. Note that only the *mget* and *update* operations are expected to be used by programmers; *restore* operations are automatically generated by the translation to the global-state semantics.

The three operations satisfy the following laws:

$$\textbf{mget-mget}: \quad mget \gg\!= (\lambda s \to mget \gg\!= k\, s) = mget \gg\!= (\lambda s \to k\, s\, s), \tag{7.2}$$

$$\textbf{update-mget}: \quad mget \gg\!= \lambda s \to update\, u \gg \eta\, (s \oplus u) = update\, u \gg mget, \tag{7.3}$$

$$\textbf{restore-mget}: \quad mget \gg\!= \lambda s \to restore\, u \gg \eta\, (s \ominus u) = restore\, u \gg mget, \tag{7.4}$$

$$\textbf{update-restore}: \quad update\, u \gg restore\, u = \eta\, (). \tag{7.5}$$

The first law for *mget* corresponds to that for *get*. The second and third law, respectively, capture the impact of *update* and *restore* on *mget*. Finally, the fourth law expresses that *restore* undoes the effect of *update*.

We can rewrite the *queens* program to make use of this *MModify* type class. Compared to the *MState*-based version in Section 2.3, we only need to replace *get* with *mget* and *put* $(s \oplus r)$ with *update r*.

```
queens_M :: (MModify (Int, [Int]) Int m, MNondet m) ⇒ Int → m [Int]
queens_M n = loop where
  loop = do (c, sol) ← mget
            if c ⩾ n then η sol
            else do r ← choose [1 . . n]
                    guard (safe r 1 sol)
                    update r
                    loop
```

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Like we did for the state effects in Section 3.1, we define a new signature $Modify_F$ representing the syntax of modification-based state effects and implement the free monad $Free\ (Modify_F\ s\ r :+: f)$ as an instance of $MModify\ s\ r$.

$$\textbf{data}\ Modify_F\ s\ r\ a = MGet\ (s \to a) \mid MUpdate\ r\ a \mid MRestore\ r\ a$$

$$\begin{aligned}
&\textbf{instance}\ (Functor\ f, Undo\ s\ r) \Rightarrow MModify\ s\ r\ (Free\ (Modify_F\ s\ r :+: f))\ \textbf{where}\\
&\quad mget \quad\ = Op\ (Inl\ (MGet\ \eta))\\
&\quad update\ r = Op\ (Inl\ (MUpdate\ r\ (\eta\ ())))\\
&\quad restore\ r = Op\ (Inl\ (MRestore\ r\ (\eta\ ())))
\end{aligned}$$

Like the h_{State} handler, the following h_{Modify} handler maps this free monad to the *StateT* monad transformer, but now using the operations $(\oplus)$ and $(\ominus)$ provided by $Undo\ s\ r$.

$$\begin{aligned}
&h_{Modify} :: (Functor\ f, Undo\ s\ r) \Rightarrow Free\ (Modify_F\ s\ r :+: f)\ a \to StateT\ s\ (Free\ f)\ a\\
&h_{Modify} = fold\ gen\ (alg \,\#\, fwd)\\
&\quad \textbf{where}\\
&\qquad gen\ x \qquad\qquad\ = StateT\ \$\ \lambda s \to \eta\ (x, s)\\
&\qquad alg\ (MGet\ k) \qquad = StateT\ \$\ \lambda s \to run_{StateT}\ (k\ s)\ s\\
&\qquad alg\ (MUpdate\ r\ k) = StateT\ \$\ \lambda s \to run_{StateT}\ k\ (s \oplus r)\\
&\qquad alg\ (MRestore\ r\ k) = StateT\ \$\ \lambda s \to run_{StateT}\ k\ (s \ominus r)\\
&\qquad fwd\ y \qquad\qquad\ = StateT\ \$\ \lambda s \to Op\ (fmap\ (\lambda k \to run_{StateT}\ k\ s)\ y)
\end{aligned}$$

It is easy to check that the four laws hold contextually up to interpretation with h_{Modify}.

Note that here we still use the *StateT* monad transformer and immutable states for the clarity of presentation and simplicity of proofs. The $(\oplus)$ and $(\ominus)$ operations also take immutable arguments. To be more efficient, we can use mutable states to implement in-place updates or use the technique of functional but in-place update (Lorenzen et al., 2023). We leave them as future work.

Similar to Sections 4.1 and 4.2, the local-state and global-state semantics of $Modify_F$ and $Nondet_F$ are given by the following functions h_{LocalM} and $h_{GlobalM}$, respectively.

$$\begin{aligned}
&h_{LocalM} :: (Functor\ f, Undo\ s\ r)\\
&\qquad \Rightarrow Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)\ a \to (s \to Free\ f\ [a])\\
&h_{LocalM} = fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}) \circ run_{StateT} \circ h_{Modify}\\
&h_{GlobalM} :: (Functor\ f, Undo\ s\ r)\\
&\qquad \Rightarrow Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)\ a \to (s \to Free\ f\ [a])\\
&h_{GlobalM} = fmap\ (fmap\ fst) \circ run_{StateT} \circ h_{Modify} \circ h_{ND+f} \circ (\Leftrightarrow)
\end{aligned}$$

For example, the locate-state interpretation of $queens_M$ is obtained through:

$$\begin{aligned}
&queens_{LocalM} :: Int \to [[Int]]\\
&queens_{LocalM} = h_{Nil} \circ flip\ h_{LocalM}\ (0, [\,]) \circ queens_M
\end{aligned}$$

7.3 Simulating local state with global state and undo

We can implement the translation from local-state semantics to global-state semantics for the modification-based state effects in a similar style to the translation *local2global* in

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Section 4.3. The translation $local2global_M$ still uses the mechanism of nondeterminism to restore previous state updates for backtracking. In Section 8, we will show a lower-level simulation of local-state semantics without relying on nondeterminism.

$$\begin{aligned}
&local2global_M :: (Functor\, f, Undo\, s\, u)\\
&\qquad\qquad \Rightarrow Free\,(Modify_F\, s\, u :+: Nondet_F :+: f)\, a\\
&\qquad\qquad \rightarrow Free\,(Modify_F\, s\, u :+: Nondet_F :+: f)\, a\\
&local2global_M = fold\, Var\, alg\\
&\quad \textbf{where}\\
&\qquad alg\,(Inl\,(MUpdate\, u\, k)) = (update\, u \,[]\, side\,(restore\, u)) \gg k\\
&\qquad alg\, p = Op\, p
\end{aligned}$$

Compared to *local2global*, the main difference is that we do not need to copy and store the whole state. Instead, we store the update u and reverse the state update using $restore\, u$ in the second branch. The following theorem shows the correctness of $local2global_M$.

Theorem 6. *Given* $Functor\, f$ *and* $Undo\, s\, u$*, the equation*

$$h_{GlobalM} \circ local2global_M = h_{LocalM}$$

holds for all programs $p :: Free\,(Modify_F\, s\, u :+: Nondet_F :+: f)\, a$ *that do not use the operation* $Op\,(Inl\, MRestore\ \_\ \_)$.

The proof of this theorem can be found in Appendix 6.

As a consequence of the theorem, we can get the desired local-state behaviour for $queens_M$ by simulating it with global-state semantics as follows:

$$\begin{aligned}
&queens_{GlobalM} :: Int \rightarrow [[Int]]\\
&queens_{GlobalM} = h_{Nil} \circ flip\, h_{GlobalM}\, (0, [\,]) \circ local2global_M \circ queens_M
\end{aligned}$$

8 Modelling local state with trail stack

In order to restore the previous state during backtracking, the approaches of Section 4.3 and Section 7 both introduce a new failing branch at every individual modification of the state. The Warren Abstract Machine (WAM) (Aït-Kaci, 1991) does this in a more efficient and lower-level way: it stores consecutive updates in a *trail stack* and then batch-processes them on backtracking. This avoids introducing any additional branches. This section first incorporates that trail-stack idea in the modification-based approach of Section 7. Then, by combining it with the earlier state-based simulation of nondeterminism, we get an overall simulation of local state in terms of two stacks, the choicepoint stack and the trail stack.

8.1 Simulating local state with global trail stack

Let us work out the trail stack idea in more detail. For that, we will need a second instance of the state effect. The primary one keeps track of the state featured in the local-state semantics. The new, secondary one keeps track of the trail stack. We can easily model this stack datastructure as a Haskell lists.

$$\textbf{newtype}\ Stack\ a = Stack\ [a]$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

We store this stack in the secondary instance of the state effect, and we add and remove elements through the *pushStack* and *popStack* functions.

```
pushStack :: MState (Stack a) m
          ⇒ a → m ()
pushStack x = do
  Stack xs ← get
  put (Stack (x : xs))
```

```
popStack :: MState (Stack a) m
         ⇒ m (Maybe a)
popStack = do
  Stack xs ← get
  case xs of
    []       → η Nothing
    (x : xs′) → do put (Stack xs′); η (Just x)
```

We store two types of entries in the trail stack. The first types are the reversible updates u (see Section 7) that we apply to the primary state. The second types are markers that mark the end of a batch on the trail stack; we represent these with the unit type (). Hence, we use the sum type *Either u* () to use both as elements of the trail stack.

When we enter a left branch, we push a *Right* () marker on the trail stack. For every state update u we perform in that branch, we push the corresponding *Left u* entry on top of the marker. When we backtrack to the right branch, we unwind the trail stack down to the marker and reverse all updates along the way. This process is known as "untrailing".

```
untrail :: (MState (Stack (Either u ())) m, MModify s u m) ⇒ m ()
untrail = do top ← popStack
             case top of
               Nothing        → η ()
               Just (Right ()) → η ()
               Just (Left u)   → restore u ≫ untrail
```

With the above trail stack functionality in place, the following translation function *local2trail* simulates the local-state semantics with global-state semantics by means of the trail stack.

```
local2trail :: (Functor f, Undo s u)
  ⇒ Free (Modify_F s u :+: Nondet_F :+: f) a
  → Free (Modify_F s u :+: Nondet_F :+: State_F (Stack (Either u ())) :+: f) a
local2trail = fold Var (alg_1 # alg_2 # fwd)
  where
    alg_1 (MUpdate u k) = pushStack (Left u) ≫ update u ≫ k
    alg_1 p             = Op ∘ Inl $ p
    alg_2 (Or p q)      = (pushStack (Right ()) ≫ p) [] (untrail ≫ q)
    alg_2 p             = Op ∘ Inr ∘ Inl $ p
    fwd p               = Op ∘ Inr ∘ Inr ∘ Inr $ p
```

As already informally explained above, this translation function 1) pushes updates to the trail tack, 2) pushes a marker to the trail stack in the left branch of a choice, and 3) untrails in the right branch. All other operations remain as is.

To ensure that *pushStack* and *popStack* access the secondary, trail-stack state in the above translation, we also need to define the following instance of *MState*.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

```
instance (Functor f, Functor g, Functor h)
  ⇒ MState s (Free (f :+: g :+: State_F s :+: h)) where
    get   = Op ∘ Inr ∘ Inr ∘ Inl $ Get η
    put x = Op ∘ Inr ∘ Inr ∘ Inl $ Put x (η ())
```

Now, we can combine the simulation *local2trail* with the global-state semantics provided by $h_{GlobalM}$, and handle the trail stack at the end.

```
h_GlobalT :: (Functor f, Undo s u)
          ⇒ Free (Modify_F s n :+: Nondet_F :+: f) a → s → Free f [a]
h_GlobalT = fmap (fmap fst ∘ flip run_StateT (Stack [ ]) ∘ h_State) ∘ h_GlobalM ∘ local2trail
```

The following theorem establishes the correctness of $h_{GlobalT}$ with respect to the local-state semantics given by h_{Local} defined in Section 4.1.

Theorem 7. *Given Functor f and Undo s u, the equation*

$$h_{GlobalT} = h_{LocalM}$$

holds for all programs $p :: Free\,(Modify_F\ s\ u :+: Nondet_F :+: f)\ a$ *that do not use the operation* $Op\,(Inl\,(MRestore\ \_\ \_))$.

The proof can be found in Appendix 7; it uses the same fold fusion strategy as in the proofs of other theorems.

8.2 Putting everything together, again

We can further combine the *local2trail* simulation with the *nondet2state* simulation of nondeterminism from Section 5 and the *state2state* simulation of multiple states from Section 6.1. The resulting simulation encodes the local-state semantics with one modification-based state and two stacks, a choicepoint stack generated by *nondet2state* and a trail stack generated by *local2trail*. This has a close relationship to the WAM of Prolog. The modification-based state models the state of the program. The choicepoint stack stores the remaining branches to implement the nondeterministic searching. The trail stack stores the previous state updates to implement the backtracking.

The combined simulation function $simulate_T$ is defined as follows:

```
simulate_T   :: (Functor f, Undo s u)
             ⇒ Free (Modify_F s u :+: Nondet_F :+: f) a
             → s → Free f [a]
simulate_T x s = extractT ∘ h_State
             ∘ fmap fst ∘ flip run_StateT s ∘ h_Modify
             ∘ (⇔) ∘ states2state ∘ (↻)
             ∘ (⇔) ∘ nondet2state ∘ (⇔)
             ∘ local2trail $ x
```

It uses the auxiliary function *extractT* to get the final results and (↻) to reorder the signatures. Note that the initial state used by *extractT* is (*SS* [ ] [ ], *Stack* [ ]), which contains an empty results list, an empty choicepoint stack, and an empty trail stack.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{array}{c}
\mathit{Free}\ (\mathit{Modify}_F\ s\ u :+: \mathit{Nondet}_F :+: f)\ a \\
\downarrow{\scriptstyle \mathit{local2trail}} \\
\mathit{Free}\ (\mathit{Modify}_F\ s\ u :+: \mathit{Nondet}_F :+: \mathit{State}_F\ (\mathit{Stack}\ (\mathit{Either}\ u\ ()))) :+: f)\ a \\
\downarrow{\scriptstyle (\Leftrightarrow) \circ \mathit{nondet2state} \circ (\Leftrightarrow)} \\
\mathit{Free}\ (\mathit{Modify}_F\ s\ u :+: \mathit{State}_F\ (\mathit{St}\ s\ u\ f\ a) :+: \mathit{State}_F\ (\mathit{Stack}\ (\mathit{Either}\ u\ ()))) :+: f)\ () \\
\downarrow{\scriptstyle (\Leftrightarrow) \circ \mathit{states2state} \circ (\circlearrowleft)} \\
\mathit{Free}\ (\mathit{Modify}_F\ s\ u :+: \mathit{State}_F\ (\mathit{St}\ s\ u\ f\ a, \mathit{Stack}\ (\mathit{Either}\ u\ ()))) :+: f)\ () \\
\downarrow{\scriptstyle \mathit{fmap}\ \mathit{fst} \circ \mathit{flip}\ \mathit{run}_{\mathit{StateT}}\ s \circ h_{\mathit{Modify}}} \\
\mathit{Free}\ (\mathit{State}_F\ (\mathit{St}\ s\ u\ f\ a, \mathit{Stack}\ (\mathit{Either}\ u\ ()))) :+: f)\ () \\
\downarrow{\scriptstyle \mathit{extractT} \circ h_{\mathit{State}}} \\
\mathit{Free}\ f\ [a]
\end{array}
$$

Fig. 5: An overview of the $\mathit{simulate}_T$ function.

$$
\begin{array}{l}
\mathit{extractT}\ x = \mathit{results}_{SS} \circ \mathit{fst} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ x\ (\mathit{SS}\ [\,]\ [\,], \mathit{Stack}\ [\,]) \\
(\circlearrowleft) :: (\mathit{Functor}\ f_1, \mathit{Functor}\ f_2, \mathit{Functor}\ f3, \mathit{Functor}\ f4) \\
\quad \Rightarrow \mathit{Free}\ (f_1 :+: f_2 :+: f3 :+: f4)\ a \to \mathit{Free}\ (f_2 :+: f3 :+: f_1 :+: f4)\ a \\
(\circlearrowleft)\ (\mathit{Var}\ x) = \mathit{Var}\ x \\
(\circlearrowleft)\ (\mathit{Op}\ (\mathit{Inl}\ k)) = (\mathit{Op} \circ \mathit{Inr} \circ \mathit{Inr} \circ \mathit{Inl})\ (\mathit{fmap}\ (\circlearrowleft)\ k) \\
(\circlearrowleft)\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ k))) = (\mathit{Op} \circ \mathit{Inr} \circ \mathit{Inl})\ (\mathit{fmap}\ (\circlearrowleft)\ k) \\
(\circlearrowleft)\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inl}\ k)))) = (\mathit{Op} \circ \mathit{Inr} \circ \mathit{Inl})\ (\mathit{fmap}\ (\circlearrowleft)\ k) \\
(\circlearrowleft)\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Inr}\ k)))) = (\mathit{Op} \circ \mathit{Inr} \circ \mathit{Inr} \circ \mathit{Inr})\ (\mathit{fmap}\ (\circlearrowleft)\ k)
\end{array}
$$

Figure 5 illustrates each step of this simulation. The state type $\mathit{St}\ s\ u\ f\ a$ is defined as $\mathit{SS}\ (\mathit{Modify}_F\ s\ u :+: \mathit{State}_F\ (\mathit{Stack}\ (\mathit{Either}\ u\ ()))) :+: f)\ a$.

In the $\mathit{simulate}_T$ function, we first use our three simulations *local2trail*, *nondet2state* and *states2state* (together with some reordering of signatures) to interpret the local-state semantics for state and nondeterminism in terms of a modification-based state and a general state containing two stacks. Then, we use the handler h_{Modify} to interpret the modification-based state effect, and use the handler h_{State} to interpret the two stacks. Finally, we use the function *extractT* to get the final results.

As in Section 6.2, we can also fuse $\mathit{simulate}_T$ into a single handler.

$$
\begin{array}{l}
\textbf{type}\ \mathit{Comp}\ f\ a\ s\ u = (\mathit{WAM}\ f\ a\ s\ u, s) \to \mathit{Free}\ f\ [a] \\
\textbf{data}\ \mathit{WAM}\ f\ a\ s\ u = \mathit{WAM}\ \{\mathit{results}\ \ :: [a] \\
\qquad\qquad\qquad\qquad\qquad\qquad , \mathit{cpStack} :: [\mathit{Comp}\ f\ a\ s\ u] \\
\qquad\qquad\qquad\qquad\qquad\qquad , \mathit{trStack}\ \ :: [\mathit{Either}\ u\ ()]\} \\
\mathit{simulate}_{TF}\ \ :: (\mathit{Functor}\ f, \mathit{Undo}\ s\ u) \\
\qquad\qquad \Rightarrow \mathit{Free}\ (\mathit{Modify}_F\ s\ u :+: \mathit{Nondet}_F :+: f)\ a \\
\qquad\qquad \to s \\
\qquad\qquad \to \mathit{Free}\ f\ [a]
\end{array}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&\mathit{simulate}_{TF}\ x\ s = \mathit{fold}\ \mathit{gen}\ (\mathit{alg}_1 \,\#\, \mathit{alg}_2 \,\#\, \mathit{fwd})\ x\ (\mathit{WAM}\ [\,]\ [\,]\ [\,], s)\\
&\quad \textbf{where}\\
&\qquad \mathit{gen}\ x \qquad\qquad (\mathit{WAM}\ xs\ cp\ tr, s) = \mathit{continue}\ (xs +\!\!+ [x])\ cp\ tr\ s\\
&\qquad \mathit{alg}_1\ (\mathit{MGet}\ k) \qquad (\mathit{WAM}\ xs\ cp\ tr, s) = k\ s\ (\mathit{WAM}\ xs\ cp\ tr, s)\\
&\qquad \mathit{alg}_1\ (\mathit{MUpdate}\ u\ k)\ (\mathit{WAM}\ xs\ cp\ tr, s) = k\ (\mathit{WAM}\ xs\ cp\ (\mathit{Left}\ u : tr), s \oplus u)\\
&\qquad \mathit{alg}_1\ (\mathit{MRestore}\ u\ k)\ (\mathit{WAM}\ xs\ cp\ tr, s) = k\ (\mathit{WAM}\ xs\ cp\ tr, s \ominus u)\\
&\qquad \mathit{alg}_2\ \mathit{Fail} \qquad (\mathit{WAM}\ xs\ cp\ tr, s) = \mathit{continue}\ xs\ cp\ tr\ s\\
&\qquad \mathit{alg}_2\ (\mathit{Or}\ p\ q) \qquad (\mathit{WAM}\ xs\ cp\ tr, s) = p\ (\mathit{WAM}\ xs\ (\mathit{untrail}\ q : cp)\ (\mathit{Right}\ () : tr), s)\\
&\qquad \mathit{fwd}\ op \qquad (\mathit{WAM}\ xs\ cp\ tr, s) = \mathit{Op}\ (\mathit{fmap}\ (\$(\mathit{WAM}\ xs\ cp\ tr, s))\ op)\\
&\qquad \mathit{untrail}\ q \qquad (\mathit{WAM}\ xs\ cp\ tr, s) = \textbf{case}\ tr\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad [\,] \to q\ (\mathit{WAM}\ xs\ cp\ tr, s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Right}\ () : tr' \to q\ (\mathit{WAM}\ xs\ cp\ tr', s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Left}\ n : tr' \quad \to \mathit{untrail}\ q\ (\mathit{WAM}\ xs\ cp\ tr', s \ominus u)\\
&\qquad \mathit{continue}\ xs\ cp\ tr\ s = \qquad \textbf{case}\ cp\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad [\,] \quad \to \eta\ xs\\
&\qquad\qquad\qquad\qquad\qquad\qquad p : cp' \to p\ (\mathit{WAM}\ xs\ cp'\ tr, s)
\end{aligned}
$$

Here, the carrier type of the algebras is $\mathit{Comp}\ f\ a\ s\ u$. It differs from that of $\mathit{simulate}_F$ in that it also takes a trail stack as an input.

N-queens with two stacks. With $\mathit{simulate}_T$, we can implement the backtracking algorithm of the n-queens problem with one modification-based state and two stacks.

$$
\begin{aligned}
&\mathit{queensSimT} :: \mathit{Int} \to [[\mathit{Int}]]\\
&\mathit{queensSimT} = h_{\mathit{Nil}} \circ \mathit{flip}\ \mathit{simulate}_T\ (0, [\,]) \circ \mathit{queens}_M
\end{aligned}
$$

9 Related work

There are various related works.

9.1 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) Warren (1983); Aït-Kaci (1991). 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 these addresses, and the process of restoring those cells is called *untrailing*.

WAM derivation and correctness. Several authors have studied the derivation of the WAM from a specification of Prolog, and its correctness.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Börger and Rosenzweig (1995) start from an operational semantics of Prolog based on derivation trees and refine this in successive steps to the WAM. Their approach was later mechanized in Isabelle/HOL by Pusch (1996). Pirog and Gibbons (2011) sketch how the WAM can be derived from a Prolog interpreter following the functional correspondence between evaluator and abstract machine Ager et al. (2005).

Neither of these approaches is based on an abstraction of effects that separates them from other aspects of Prolog.

The 4-port box model. While trailing happens under the hood, there is a folklore Prolog programming pattern for observing and intervening at different point 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:

```
(putStr "call" [] side (putStr "fail")) >>
p >>= λx →
(putStr "exit" [] side (putStr "redo")) >>
η x
```

This technique was applied in the monadic setting by Hinze (1996), and it has been our inspiration for expressing the state restoration with global state.

Functional models of prolog. Various authors have modelled (aspects of) Prolog in functional programming languages, often using monads to capture nondeterminism and state effects. Notably, Spivey and Seres (1999) develop an embedding of Prolog in Haskell.

Most attention has gone towards modelling the nondeterminism or search aspect of Prolog, with various monads and monad transformers being proposed (Hinze, 2000; Kiselyov et al., 2005). Notably, Schrijvers et al. (2014) shows how Prolog's search can be exposed with a free monad and manipulated using handlers.

None of these works consider mapping high-level to low-level representations of the effects.

9.2 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. Gibbons and Hinze (2011) proposed to reason axiomatically about programs with effects and provided an axiomatic characterization of local state semantics. Our earlier work in Pauwels et al. (2019) was directly inspired by their work: we introduced an axiomatic characterization of global state and used axiomatic reasoning to prove handling local state with global state correct. We also provided models that satisfy the axioms, whereas their paper mistakenly claims that one model satisfies the local state axioms and that another model is monadic. This paper is an extension of Pauwels et al.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

(2019), but notably, we depart from the axiomatic reasoning approach; instead we use proof techniques based on algebraic effects and handlers.

Algebraic effects. Our formulation of implementing local state with global state is directly inspired by the effect handlers approach of Plotkin and Pretnar (2009). By making the free monad explicit our proofs benefit directly from the induction principle that Bauer and Pretnar established for effect handler programs . 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, Lukšič and Pretnar (2020) 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 allows expressing at the type level that a handler reduces a higher-level effect to a lower-level one.

Wu and Schrijvers (2015) first presented fusion as a technique for optimizing compositions of effect handlers. They use a specific form of fusion known as fold–build fusion or short-cut fusion (Gill et al., 1993). To enable this kind of fusion they transform the handler algebras to use the codensity monad as their carrier. Their approach is not directly usable because it does not fuse non-handler functions, and we derive simpler algebras (not obfuscated by the codensity monad) than those they do.

More recently, Yang and Wu (2021) have used the fusion approach of Wu and Schrijvers (2015) (but with the continuation monad rather than the codensity monad) for reasoning; they remark that, although handlers are composable, the semantics of these composed handlers are not always obvious and that determining the correct order of composition to arrive at a desired semantics is nontrivial. They propose a technique based on modular handlers (Schrijvers et al., 2019), which considers conditions under which the fusion of these modular handlers respect not only the laws of each of the handler's algebraic theories but also additional interaction laws. Using this technique they provide succinct proofs of the correctness of local state handlers, constructed from a fusion of state and nondeterminism handlers.

Earlier versions. This paper refines and much expands on two earlier works of the last author.

Pauwels et al. (2019) have the same goal as Section 4: it uses the state-restoring version of put to simulate local state with global state. It differs from this work in that it relies on an axiomatic (i.e., law-based), as opposed to handler-based, semantics for local and global state. This means that handler fusion cannot be used as a reasoning technique. Moreover, it uses a rather heavy-handed syntactic approach to contextual equivalence, and it assumes that no other effects are invoked.

Another precursor is the work of Seynaeve et al. (2020), which establishes similar results as those in Section 5.1. However, instead of generic definitions for the free monad and its fold, they use a specialized free monad for nondeterminism and ordinary recursive functions for handling. As a consequence, their proofs use structural induction rather than fold fusion. Furthermore, they did not consider other effects either.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

10 Conclusion and future work

We studied the simulations of higher-level effects with lower-level effects for state and nondeterminism. We started with the translation from the local-state semantics of state and nondeterminism to the global-state semantics. Then, we further showed how to translate nondeterminism to state (a choicepoint stack), and translate multiple state effects into one state effect. Combining these results, we can simulate the local-state semantics, a high-level programming abstraction, with only one low-level state effect. We also demonstrated that we can simulate the local-state semantics using a trail stack in a similar style to the Warren Abstract Machine of Prolog. We define the effects and their translations with algebraic effects and effect handlers, respectively. These are implemented as free monads and folds in Haskell. The correctness of all these translations has been proved using the technique of program calculation, especially using the fusion properties.

In future work, we would like to explore the potential optimizations enabled by mutable states. Mutable states fit the global-state semantics naturally. With mutable states, we can implement more efficient state update and restoration operations for the simulation $local2global_M$ (Section 7), as well as more efficient implementations of the choicepoint stacks and trail stacks used by the simulations *nondet2state* (Section 5.2) and *local2trail* (Section 8), respectively. We would also like to consider the low-level simulations of other control-flow constructs used in logical programming languages such as Prolog's *cut* operator for trimming the search space. Since operators like *cut* are usually implemented as scoped or higher order effects (Piróg et al., 2018; Wu et al., 2014; Yang et al., 2022; van den Berg and Schrijvers, 2023), we would have to adapt our approach accordingly.

Conflicts of Interest

None.

References

Ager, M. S., Danvy, O. & Midtgaard, J. (2005) A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.* **342**(1), 149–172. Applied Semantics: Selected Topics.

Aït-Kaci, H. (1991) *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.

Börger, E. & Rosenzweig, D. (1995) The WAM – definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence*, Beierle, C. & Plümer, L. (eds). Elsevier Science B.V./North-Holland, pp. 20–90.

Felleisen, M. (1991) On the expressive power of programming languages. *Sci. Comput. Program.* **17**(1–3), 35–75.

Gale, Y. (2007) ListT done right alternative. Available at: `https://wiki.haskell.org/ListT_done_right_alternative`.

Gibbons, J. (2000) Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000*, Revised Lectures. Springer, pp. 149–202.

Gibbons, J. & Hinze, R. (2011) Just do it: Simple monadic equational reasoning. *SIGPLAN Not.* **46**(9), 2–14.

Gill, A., Launchbury, J. & Peyton Jones, S. L. (1993) A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. New York, NY, USA. Association for Computing Machinery, pp. 223–232.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Hinze, R. (1996) Monadic-style backtracking. Technical Report IAI-TR-96-9. Institut für Informatik III, Universität Bonn.

Hinze, R. (2000) Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, Montreal, Canada, September 18-21, 2000. ACM. pp. 186–197.

Hutton, G. & Fulger, D. (2008) *Reasoning about Effects: Seeing the Wood through the Trees (Extended Version)*.

Jones, M. P. (1995) Functional programming with overloading and higher-order polymorphism.

Kiselyov, O. (2015) Laws of monadplus. Available at: http://okmij.org/ftp/Computation/monads.html#monadplus.

Kiselyov, O. & Ishii, H. (2015) Freer monads, more extensible effects. *SIGPLAN Not.* **50**(12), 94–105.

Kiselyov, O., Shan, C., Friedman, D. P. & Sabry, A. (2005) Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, Tallinn, Estonia, September 26-28, 2005. ACM, pp. 192–203.

Lorenzen, A., Leijen, D. & Swierstra, W. (2023) FP^2: Fully in-place functional programming. *Proc. ACM Program. Lang.* **7**(ICFP).

Lukšič, Z. & Pretnar, M. (2020) Local algebraic effect theories. *J. Funct. Program.* **30**, e13.

Moggi, E. (1991) Notions of computation and monads. *Inform. Computat*. **93**(1), 55–92.

Pauwels, K., Schrijvers, T. & Mu, S.-C. (2019) Handling local state with global state. In *International Conference on Mathematics of Program Construction*. Springer, pp. 18–44.

Pirog, M. & Gibbons, J. (2011) A functional derivation of the warren abstract machine. Unpublished.

Piróg, M., Schrijvers, T., Wu, N. & Jaskelioff, M. (2018) Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, Oxford, UK, July 09-12, 2018, pp. 809–818.

Plotkin, G. & Pretnar, M. (2009) Handlers of algebraic effects. In *Programming Languages and Systems*. Berlin, Heidelberg. Springer Berlin Heidelberg, pp. 80–94.

Plotkin, G. D. & Power, J. (2002) Notions of computation determine monads. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*. Berlin, Heidelberg. Springer-Verlag, pp. 342–356.

Plotkin, G. D. & Power, J. (2003) Algebraic operations and generic effects. *Appl. Categorical Struct.* **11**(1), 69–94.

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

Pusch, C. (1996) Verification of compiler correctness for the wam. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*. Berlin, Heidelberg. Springer-Verlag, pp. 347–361.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, Paris, France, September 19-23, 1983. North-Holland/IFIP. pp. 513–523.

Rivas, E., Jaskelioff, M. & Schrijvers, T. (2018) A unified view of monadic and applicative nondeterminism. *Sci. Comput. Program.* **152**, 70–98.

Schrijvers, T., Piróg, M., Wu, N. & Jaskelioff, M. (2019) Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell, ICFP 2019*, Berlin, Germany, August 18-23, 2019. pp. 98–113.

Schrijvers, T., Wu, N., Desouter, B. & Demoen, B. (2014) Heuristics entwined with handlers combined: From functional specification to logic programming implementation. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, Kent, Canterbury, United Kingdom, September 8-10, 2014. ACM, pp. 259–270.

Schulte, C. (1999) Comparing trailing and copying for constraint programming. In *Proceedings of the Sixteenth International Conference on Logic Programming*, Las Cruces, NM, USA. The MIT Press, pp. 275–289.

Seynaeve, W., Pauwels, K. & Schrijvers, T. (2020) State will do. In *International Symposium on Trends in Functional Programming*. Springer, pp. 204–225.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Spivey, J. M. & Seres, S. (1999) Embedding Prolog in Haskell.
Swierstra, W. (2008) Data types à la carte. *J. Funct. Program.* **18**(4), 423–436.
van den Berg, B. & Schrijvers, T. (2023) A framework for higher-order effects & handlers. *CoRR*. abs/2302.01415.
Voigtländer, J. (2009) Free theorems involving type constructor classes: functional pearl. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009*, Edinburgh, Scotland, UK, August 31 - September 2, 2009. ACM, pp. 173–184.
Volkov, N. (2014) list-t: ListT done right. https://github.com/nikita-volkov/list-t.
Wadler, P. (1989) Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989*, London, UK, September 11-13, 1989. ACM, pp. 347–359.
Warren, D. H. D. (1983) An abstract prolog instruction set. Technical Report 309. AI Center, SRI International. 333 Ravenswood Ave., Menlo Park, CA 94025.
Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94.
Wu, N. & Schrijvers, T. (2015) Fusion for free: Efficient algebraic effect handlers. In *MPC 2015*.
Wu, N., Schrijvers, T. & Hinze, R. (2014) Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Gothenburg, Sweden, September 4-5, 2014. pp. 1–12.
Yang, Z., Paviotti, M., Wu, N., van den Berg, B. & Schrijvers, T. (2022) Structured handling of scoped effects. In *Programming Languages and Systems – 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022*, Munich, Germany, April 2-7, 2022, Proceedings. Springer, pp. 462–491.
Yang, Z. & Wu, N. (2021) Reasoning about effect interaction by fusion. *Proc. ACM Program. Lang.* **5**(ICFP), 1–29.

1 Proofs for get laws in local-state semantics

In this section, we prove two equations about the interaction of nondeterminism and state in the local-state semantics.

Equation (4.3): $get \gg \varnothing = \varnothing$

Proof

$$
\begin{aligned}
& get \gg \varnothing \\
= \quad & \{\text{- definition of } (\gg) \text{ -}\} \\
& get \gg\!= (\lambda s \to \varnothing) \\
= \quad & \{\text{- Law (4.1): put-right-identity -}\} \\
& get \gg\!= (\lambda s \to put\ s \gg \varnothing) \\
= \quad & \{\text{- Law (2.5): associativity of } (\gg) \text{ -}\} \\
& (get \gg\!= put) \gg \varnothing \\
= \quad & \{\text{- Law (2.12): get-put -}\} \\
& \eta\,() \gg \varnothing \\
= \quad & \{\text{- Law (2.3): return-bind and definition of } (\gg) \text{ -}\} \\
& \varnothing
\end{aligned}
$$

■

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Equation (4.4): $get \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x) = (get \gg\!= k_1) \,[\!]\, (get \gg\!= k_2)$

Proof

$$
\begin{aligned}
& get \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x) \\
= \;& \{\text{- Law (2.3): return-bind and definition of } (\gg) \text{ -}\} \\
& \eta\, () \gg (get \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x)) \\
= \;& \{\text{- Law (2.5): associativity of } (\gg\!=) \text{ -}\} \\
& (\eta\, () \gg get) \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x) \\
= \;& \{\text{- Law (2.12): get-put -}\} \\
& ((get \gg\!= put) \gg get) \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x) \\
= \;& \{\text{- Law (2.5): associativity of } (\gg) \text{ -}\} \\
& (get \gg\!= (\lambda s \to put\, s \gg get)) \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x) \\
= \;& \{\text{- Law (2.5): associativity of } (\gg\!=) \text{ -}\} \\
& get \gg\!= (\lambda s \to (\lambda s \to put\, s \gg get)\, s \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x)) \\
= \;& \{\text{- function application -}\} \\
& get \gg\!= (\lambda s \to (put\, s \gg get) \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x)) \\
= \;& \{\text{- Law (2.11): put-get -}\} \\
& get \gg\!= (\lambda s \to (put\, s \gg \eta\, s) \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x)) \\
= \;& \{\text{- Law (2.5): associativity of } (\gg) \text{ -}\} \\
& get \gg\!= (\lambda s \to put\, s \gg (\eta\, s \gg\!= (\lambda x \to k_1\, x \,[\!]\, k_2\, x))) \\
= \;& \{\text{- Law (2.3): return-bind and function application -}\} \\
& get \gg\!= (\lambda s \to put\, s \gg (k_1\, s \,[\!]\, k_2\, s)) \\
= \;& \{\text{- Law (4.2): put-left-distributivity -}\} \\
& get \gg\!= (\lambda s \to (put\, s \gg k_1\, s) \,[\!]\, (put\, s \gg k_2\, s)) \\
= \;& \{\text{- Law (2.3): return-bind (twice) -}\} \\
& get \gg\!= (\lambda s \to (put\, s \gg (\eta\, s \gg\!= k_1)) \,[\!]\, (put\, s \gg (\eta\, s \gg\!= k_2))) \\
= \;& \{\text{- Law (2.5): associativity of } (\gg) \text{ -}\} \\
& get \gg\!= (\lambda s \to ((put\, s \gg \eta\, s) \gg\!= k_1) \,[\!]\, ((put\, s \gg \eta\, s) \gg\!= k_2)) \\
= \;& \{\text{- Law (2.11): put-get -}\} \\
& get \gg\!= (\lambda s \to ((put\, s \gg get) \gg\!= k_1) \,[\!]\, ((put\, s \gg get) \gg\!= k_2)) \\
= \;& \{\text{- Law (2.5): associativity of } (\gg) \text{ -}\} \\
& get \gg\!= (\lambda s \to (put\, s \gg (get \gg\!= k_1)) \,[\!]\, (put\, s \gg (get \gg\!= k_2))) \\
= \;& \{\text{- Law (4.2): put-left-distributivity -}\} \\
& get \gg\!= (\lambda s \to put\, s \gg ((get \gg\!= k_1) \,[\!]\, (get \gg\!= k_2))) \\
= \;& \{\text{- Law (2.5): associativity of } (\gg\!=) \text{ -}\} \\
& (get \gg\!= put) \gg ((get \gg\!= k_1) \,[\!]\, (get \gg\!= k_2)) \\
= \;& \{\text{- Law (2.12): get-put -}\} \\
& \eta\, () \gg ((get \gg\!= k_1) \,[\!]\, (get \gg\!= k_2)) \\
= \;& \{\text{- Law (2.3): return-bind and definition of } (\gg) \text{ -}\} \\
& (get \gg\!= k_1) \,[\!]\, (get \gg\!= k_2)
\end{aligned}
$$

■

2 Proofs for modelling local state with global state

This section proves the following theorem in Section 4.3.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Theorem 1. $h_{Global} \circ local2global = h_{Local}$

Preliminary. It is easy to see that $run_{StateT} \circ h_{State}$ can be fused into a single fold defined as follows:

$$
\begin{aligned}
&h_{State1} :: Functor\, f \Rightarrow Free\,(State_F\, s :+: f)\, a \to (s \to Free\, f\, (a, s)) \\
&h_{State1} = fold\, gen_S\, (alg_S \,\#\, fwd_S) \\
&\quad \textbf{where} \\
&\qquad gen_S\, x \quad\quad\; s = Var\,(x, s) \\
&\qquad alg_S\,(Get\, k) \;\; s = k\, s\, s \\
&\qquad alg_S\,(Put\, s\, k)\; \_ = k\, s \\
&\qquad fwd_S\, y \quad\quad\; s = Op\,(fmap\,(\$s)\, y)
\end{aligned}
$$

For brevity, we use h_{State1} to replace $run_{StateT} \circ h_{State}$ in the following proofs.

2.1 Main proof structure

The main theorem we prove in this section is

Theorem 8. $h_{Global} \circ local2global = h_{Local}$

Proof Both the left-hand side and the right-hand side of the equation consist of function compositions involving one or more folds. We apply fold fusion separately on both sides to contract each into a single fold:

$$
\begin{aligned}
h_{Global} \circ local2global &= fold\, gen_{LHS}\, (alg^S_{LHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{LHS}) \\
h_{Local} &= fold\, gen_{RHS}\, (alg^S_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS})
\end{aligned}
$$

We approach this calculationally. That is to say, we do not first postulate definitions of the unknowns above (alg^S_{LHS} and so on) and then verify whether the fusion conditions are satisfied. Instead, we discover the definitions of the unknowns. We start from the known side of each fusion condition and perform case analysis on the possible shapes of input. By simplifying the resulting case-specific expression, and pushing the handler applications inwards, we end up at a point where we can read off the definition of the unknown that makes the fusion condition hold for that case.

Finally, we show that both folds are equal by showing that their corresponding parameters are equal:

$$
\begin{aligned}
gen_{LHS} &= gen_{RHS} \\
alg^S_{LHS} &= alg^S_{RHS} \\
alg^{ND}_{LHS} &= alg^{ND}_{RHS} \\
fwd_{LHS} &= fwd_{RHS}
\end{aligned}
$$

A noteworthy observation is that, for fusing the left-hand side of the equation, we do not use the standard fusion rule:

$$
\begin{aligned}
h_{Global} \circ fold\, Var\, alg &= fold\, (h_{Global} \circ Var)\, alg' \\
\Leftarrow \quad h_{Global} \circ alg &= alg' \circ fmap\, h_{Global}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

where $local2global = fold\ Var\ alg$. The problem is that we will not find an appropriate alg' such that $alg'\ (fmap\ h_{Global}\ t)$ restores the state for any t of type $(State_F\ s :+: Nondet_F :+: f)\ (Free\ (State_F\ s :+: NonDetF :+: f)\ a)$.

Fortunately, we do not need such an alg'. As we have already pointed out, we can assume that the subterms of t have already been transformed by *local2global*, and thus all occurrences of *Put* appear in the put_R constellation.

We can incorporate this assumption by using the alternative fusion rule:

$$\begin{array}{rrcl} & h_{Global} \circ fold\ Var\ alg & = & fold\ (h_{Global} \circ Var)\ alg' \\ \Leftarrow & h_{Global} \circ alg \circ fmap\ local2global & = & alg' \circ fmap\ h_{Global} \circ fmap\ local2global \end{array}$$

The additional $fmap\ local2global$ in the condition captures the property that all the subterms have been transformed by *local2global*.

In order to not clutter the proofs, we abstract everywhere over this additional $fmap\ local2global$ application, except in the one place where we need it. That is the appeal to the key lemma:

$$\begin{array}{c} h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ t)))\ s \\ = \\ \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ t)))\ s;\ \eta\ (x, s) \end{array}$$

This expresses that the syntactic transformation *local2global* makes sure that, despite any temporary changes, the computation t restores the state back to its initial value.

We elaborate each of these steps below. ■

2.2 Fusing the right-hand side

We calculate as follows:

$$\begin{array}{ll} & h_{Local} \\ = & \{\text{- definition -}\} \\ & h_L \circ h_{State1} \\ & \quad \text{with} \\ & \quad\quad h_L :: (Functor\ f) \Rightarrow (s \to Free\ (Nondet_F :+: f)\ (a, s)) \to s \to Free\ f\ [a] \\ & \quad\quad h_L = fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}) \\ = & \{\text{- definition of } h_{State1} \text{ -}\} \\ & h_L \circ fold\ gen_S\ (alg_S \,\#\, fwd_S) \\ = & \{\text{- fold fusion-post (Equation 3.2) -}\} \\ & fold\ gen_{RHS}\ (alg^S_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS}) \end{array}$$

This last step is valid provided that the fusion conditions are satisfied:

$$\begin{array}{rcl} h_L \circ gen_S & = & gen_{RHS} \\ h_L \circ (alg_S \,\#\, fwd_S) & = & (alg^S_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS}) \circ fmap\ h_L \end{array}$$

We calculate for the first fusion condition:

$$\begin{array}{ll} & h_L\ (gen_S\ x) \\ = & \{\text{- definition of } gen_S \text{ -}\} \end{array}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_L\ (\lambda s \to Var\ (x, s)) \\
= & \quad \{\text{- definition of } h_L \text{ -}\} \\
& fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to Var\ (x, s)) \\
= & \quad \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (Var\ (x, s))) \\
= & \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (Var\ [(x, s)]) \\
= & \quad \{\text{- definition of } fmap \text{ (twice) -}\} \\
& \lambda s \to Var\ [x] \\
= & \quad \{\text{- define } gen_{RHS}\ x = \lambda s \to Var\ [x] \text{ -}\} \\
= & \ gen_{RHS}\ x
\end{aligned}
$$

We conclude that the first fusion condition is satisfied by

$$
\begin{aligned}
& gen_{RHS} :: Functor\ f \Rightarrow a \to (s \to Free\ f\ [a]) \\
& gen_{RHS}\ x = \lambda s \to Var\ [x]
\end{aligned}
$$

The second fusion condition decomposes into two separate conditions:

$$
\begin{aligned}
h_L \circ alg_S &= alg^S_{RHS} \circ fmap\ h_L \\
h_L \circ fwd_S &= (alg^{ND}_{RHS}\ \#\ fwd_{RHS}) \circ fmap\ h_L
\end{aligned}
$$

We calculate for the first subcondition:

case $t = Get\ k$

$$
\begin{aligned}
& h_L\ (alg_S\ (Get\ k)) \\
= & \quad \{\text{- definition of } alg_S \text{ -}\} \\
& h_L\ (\lambda s \to k\ s\ s) \\
= & \quad \{\text{- definition of } h_L \text{ -}\} \\
& fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to k\ s\ s) \\
= & \quad \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ s\ s)) \\
= & \quad \{\text{- beta-expansion (twice) -}\} \\
= & \ \lambda s \to (\lambda s_1\ s_2 \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ s_2\ s_1)))\ s\ s \\
= & \quad \{\text{- definition of } fmap \text{ (twice) -}\} \\
= & \ \lambda s \to (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (\lambda s_1\ s_2 \to k\ s_2\ s_1))\ s\ s \\
= & \quad \{\text{- eta-expansion of } k \text{ -}\} \\
= & \ \lambda s \to (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ k)\ s\ s \\
= & \quad \{\text{- define } alg^S_{RHS}\ (Get\ k) = \lambda s \to k\ s\ s \text{ -}\} \\
= & \ alg^S_{RHS}\ (Get\ (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ k)) \\
= & \quad \{\text{- definition of } fmap \text{ -}\} \\
= & \ alg^S_{RHS}\ (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (Get\ k)) \\
= & \quad \{\text{- definition of } h_L \text{ -}\} \\
= & \ alg^S_{RHS}\ (fmap\ h_L\ (Get\ k))
\end{aligned}
$$

case $t = Put\ s\ k$

$$
\begin{aligned}
& h_L\ (alg_S\ (Put\ s\ k)) \\
= & \quad \{\text{- definition of } alg_S \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_L\,(\lambda\_ \to k\,s) \\
= \;& \{\text{- definition of } h_L \text{ -}\} \\
& fmap\,(fmap\,(fmap\,fst) \circ h_{ND+f})\,(\lambda\_ \to k\,s) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& \lambda\_ \to fmap\,(fmap\,fst)\,(h_{ND+f}\,(k\,s)) \\
= \;& \{\text{- beta-expansion -}\} \\
= \;& \lambda\_ \to (\lambda s_1 \to fmap\,(fmap\,fst)\,(h_{ND+f}\,(k\,s_1)))\,s \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
= \;& \lambda\_ \to (fmap\,(fmap\,(fmap\,fst) \circ h_{ND+f})\,(\lambda s_1 \to k\,s_1))\,s \\
= \;& \{\text{- eta-expansion of } k \text{ -}\} \\
= \;& \lambda\_ \to (fmap\,(fmap\,(fmap\,fst) \circ h_{ND+f})\,k)\,s \\
= \;& \{\text{- define } alg^S_{RHS}\,(Pus\,s\,k) = \backslash\_ \to k\,s \text{ -}\} \\
= \;& alg^S_{RHS}\,(Put\,s\,(fmap\,(fmap\,(fmap\,fst) \circ h_{ND+f})\,k)) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
= \;& alg^S_{RHS}\,(fmap\,(fmap\,(fmap\,fst) \circ h_{ND+f}))\,(Put\,s\,k)) \\
= \;& \{\text{- definition of } h_L \text{ -}\} \\
= \;& alg^S_{RHS}\,(fmap\,h_L\,(Put\,s\,k))
\end{aligned}
$$

We conclude that the first subcondition is met by taking:

$$
\begin{aligned}
& alg^S_{RHS} :: Functor\,f \Rightarrow State_F\,s\,(s \to Free\,f\,[a]) \to (s \to Free\,f\,[a]) \\
& alg^S_{RHS}\,(Get\,k) \;\; = \lambda s \to k\,s\,s \\
& alg^S_{RHS}\,(Put\,s\,k) = \lambda\_ \to k\,s
\end{aligned}
$$

The second subcondition can be split up in two further subconditions:

$$
\begin{aligned}
h_L \circ fwd_S \circ Inl &= alg^{ND}_{RHS} \circ fmap\,h_L \\
h_L \circ fwd_S \circ Inr &= fwd_{RHS} \circ fmap\,h_L
\end{aligned}
$$

For the first of these, we calculate:

$$
\begin{aligned}
& h_L\,(fwd_S\,(Inl\,op)) \\
= \;& \{\text{- definition of } fwd_S \text{ -}\} \\
& h_L\,(\lambda s \to Op\,(fmap\,(\$s)\,(Inl\,op))) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& h_L\,(\lambda s \to Op\,(Inl\,(fmap\,(\$s)\,op))) \\
= \;& \{\text{- definition of } h_L \text{ -}\} \\
& fmap\,(fmap\,(fmap\,fst) \circ h_{ND+f})\,(\lambda s \to Op\,(Inl\,(fmap\,(\$s)\,op))) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\,(fmap\,fst)\,(h_{ND+f}\,(Op\,(Inl\,(fmap\,(\$s)\,op)))) \\
= \;& \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\,(fmap\,fst)\,(alg_{ND+f}\,(fmap\,h_{ND+f}\,(fmap\,(\$s)\,op)))
\end{aligned}
$$

We split on *op*:
case $op = Fail$

$$
\begin{aligned}
& \lambda s \to fmap\,(fmap\,fst)\,(alg_{ND+f}\,(fmap\,h_{ND+f}\,(fmap\,(\$s)\,Fail))) \\
= \;& \{\text{- defintion of } fmap \text{ (twice) -}\} \\
& \lambda s \to fmap\,(fmap\,fst)\,(alg_{ND+f}\,Fail)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } alg_{ND+f} \text{ -}\} \\
&\quad \lambda s \to fmap\ (fmap\ fst)\ (Var\ [\,]) \\
&= \quad \{\text{- definition of } fmap \text{ (twice) -}\} \\
&\quad \lambda s \to Var\ [\,] \\
&= \quad \{\text{- define } alg^{ND}_{RHS}\ Fail = \lambda s \to Var\ [\,] \text{ -}\} \\
&\quad alg^{ND}_{RHS}\ Fail \\
&= \quad \{\text{-definition fo } fmap \text{ -}\} \\
&\quad alg^{ND}_{RHS}\ (fmap\ h_L\ fail)
\end{aligned}
$$

case $op = Or\ p\ q$

$$
\begin{aligned}
&\quad \lambda s \to fmap\ (fmap\ fst)\ (alg_{ND+f}\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ (Or\ p\ q)))) \\
&= \quad \{\text{- defintion of } fmap \text{ (twice) -}\} \\
&\quad \lambda s \to fmap\ (fmap\ fst)\ (alg_{ND+f}\ (Or\ (h_{ND+f}\ (p\ s))\ (h_{ND+f}\ (q\ s)))) \\
&= \quad \{\text{- definition of } alg_{ND+f} \text{ -}\} \\
&\quad \lambda s \to fmap\ (fmap\ fst)\ (liftM2\ (+\!\!+)\ (h_{ND+f}\ (p\ s))\ (h_{ND+f}\ (q\ s))) \\
&= \quad \{\text{- Lemma 3 -}\} \\
&\quad \lambda s \to liftM2\ (+\!\!+)\ (fmap\ (fmap\ fst)\ (h_{ND+f}\ (p\ s)))\ (fmap\ (fmap\ fst)\ (h_{ND+f}\ (q\ s))) \\
&= \quad \{\text{- define } alg^{ND}_{RHS}\ (Or\ p\ q) = \lambda s \to liftM2\ (+\!\!+)\ (p\ s)\ (q\ s) \text{ -}\} \\
&\quad alg^{ND}_{RHS}\ (Or\ (fmap\ (fmap\ fst) \circ h_{ND+f} \circ p)\ (fmap\ (fmap\ fst) \circ h_{ND+f} \circ q)) \\
&= \quad \{\text{- defintion of } fmap \text{ (twice) -}\} \\
&\quad alg^{ND}_{RHS}\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (Or\ p\ q)) \\
&= \quad \{\text{- defintion of } h_L \text{ -}\} \\
&\quad alg^{ND}_{RHS}\ (fmap\ h_L\ (Or\ p\ q))
\end{aligned}
$$

From this we conclude that the definition of alg^{ND}_{RHS} should be:

$$
\begin{aligned}
&alg^{ND}_{RHS} :: Functor\ f \Rightarrow Nondet_F\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a]) \\
&alg^{ND}_{RHS}\ Fail \quad\quad = \lambda s \to Var\ [\,] \\
&alg^{ND}_{RHS}\ (Or\ p\ q) = \lambda s \to liftM2\ (+\!\!+)\ (p\ s)\ (q\ s)
\end{aligned}
$$

For the last subcondition, we calculate:

$$
\begin{aligned}
&\quad h_L\ (fwd_S\ (Inr\ op)) \\
&= \quad \{\text{- definition of } fwd_S \text{ -}\} \\
&\quad h_L\ (\lambda s \to Op\ (fmap\ (\$s)\ (Inr\ op))) \\
&= \quad \{\text{- definition of } fmap \text{ -}\} \\
&\quad h_L\ (\lambda s \to Op\ (Inr\ (fmap\ (\$s)\ op))) \\
&= \quad \{\text{- definition of } h_L \text{ -}\} \\
&\quad fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to Op\ (Inr\ (fmap\ (\$s)\ op))) \\
&= \quad \{\text{- definition of } fmap \text{ -}\} \\
&\quad \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (Op\ (Inr\ (fmap\ (\$s)\ op)))) \\
&= \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
&\quad \lambda s \to fmap\ (fmap\ fst)\ (fwd_{ND+f}\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ op))) \\
&= \quad \{\text{- definition of } fwd_{ND+f} \text{ -}\} \\
&\quad \lambda s \to fmap\ (fmap\ fst)\ (Op\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ op))) \\
&= \quad \{\text{- definition of } fmap \text{ -}\} \\
&\quad \lambda s \to Op\ (fmap\ (fmap\ (fmap\ fst))\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ op)))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{array}{ll}
= & \{\text{- definition of } h_L \text{ -}\} \quad | \\
& \lambda s \rightarrow Op\,(h_L\,(fmap\,(\$s)\,op)) \\
= & \{\text{- Lemma 2 -}\} \\
& \lambda s \rightarrow Op\,(fmap\,(\$s)\,(fmap\,h_L\,op)) \\
= & \{\text{- define } fwd_{RHS}\,op = \lambda s \rightarrow Op\,(fmap\,(\$s)\,op) \text{ -}\} \\
& fwd_{RHS}\,(fmap\,h_L\,op)
\end{array}
$$

From this we conclude that the definition of fwd_{RHS} should be:

$$
\begin{array}{l}
fwd_{RHS} :: Functor\,f \Rightarrow f\,(s \rightarrow Free\,f\,[a]) \rightarrow (s \rightarrow Free\,f\,[a]) \\
fwd_{RHS}\,op = \lambda s \rightarrow Op\,(fmap\,(\$s)\,op)
\end{array}
$$

2.3 Fusing the left-hand side

We proceed in the same fashion with fusing left-hand side, discovering the definitions that we need to satisfy the fusion condition.

We calculate as follows:

$$
\begin{array}{ll}
& h_{Global} \circ local2global \\
= & \{\text{- definition of } local2global \text{ -}\} \\
& h_{Global} \circ fold\,Var\,alg \\
& \quad \textbf{where} \\
& \quad\quad alg\,(Inl\,(Put\,t\,k)) = put_R\,t \gg k \\
& \quad\quad alg\,p = Op\,p \\
= & \{\text{- fold fusion-post (Equation 3.2) -}\} \\
& fold\,gen_{LHS}\,(alg^S_{LHS} \,\#\, alg^{ND}_{LHS} \,\#\, fwd_{LHS})
\end{array}
$$

This last step is valid provided that the fusion conditions are satisfied:

$$
\begin{array}{rcl}
h_{Global} \circ Var & = & gen_{LHS} \\
h_{Global} \circ alg & = & (alg^S_{LHS} \,\#\, alg^{ND}_{LHS} \,\#\, fwd_{LHS}) \circ fmap\,h_{Global}
\end{array}
$$

We calculate for the first fusion condition:

$$
\begin{array}{ll}
& h_{Global}\,(Var\,x) \\
= & \{\text{- definition of } h_{Global} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(Var\,x)))) \\
= & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{State1}\,(h_{ND+f}\,(Var\,x))) \\
= & \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{State1}\,(Var\,[x])) \\
= & \{\text{- definition of } h_{State1} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(\lambda s \rightarrow Var\,([x], s)) \\
= & \{\text{- definition of } fmap \text{ (twice) -}\} \\
& \lambda s \rightarrow Var\,[x] \\
= & \{\text{- define } gen_{LHS}\,x = \lambda s \rightarrow Var\,[x] \text{ -}\} \\
& gen_{LHS}\,x
\end{array}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

We conclude that the first fusion condition is satisfied by

$$gen_{LHS} :: Functor\, f \Rightarrow a \to (s \to Free\, f\, [a])$$
$$gen_{LHS}\, x = \lambda s \to Var\, [x]$$

We can split the second fusion condition in three subconditions:

$$\begin{aligned} h_{Global} \circ alg \circ Inl &= alg^{S}_{LHS} \circ fmap\, h_{Global} \\ h_{Global} \circ alg \circ Inr \circ Inl &= alg^{ND}_{LHS} \circ fmap\, h_{Global} \\ h_{Global} \circ alg \circ Inr \circ Inr &= fwd_{LHS} \circ fmap\, h_{Global} \end{aligned}$$

Let's consider the first subconditions. It has two cases:

case $op = Get\, k$

$$\begin{aligned} & h_{Global}\, (alg\, (Inl\, (Get\, k))) \\ = \;& \{\text{- definition of } alg \text{ -}\} \\ & h_{Global}\, (Op\, (Inl\, (Get\, k))) \\ = \;& \{\text{- definition of } h_{Global} \text{ -}\} \\ & fmap\, (fmap\, fst)\, (h_{State1}\, (h_{ND+f}\, ((\Leftrightarrow)\, (Op\, (Inl\, (Get\, k)))))) \\ = \;& \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\ & fmap\, (fmap\, fst)\, (h_{State1}\, (h_{ND+f}\, (Op\, (Inr\, (Inl\, (fmap\, (\Leftrightarrow)\, (Get\, k))))))) \\ = \;& \{\text{- definition of } fmap \text{ -}\} \\ & fmap\, (fmap\, fst)\, (h_{State1}\, (h_{ND+f}\, (Op\, (Inr\, (Inl\, (Get\, ((\Leftrightarrow) \circ k))))))) \\ = \;& \{\text{- definition of } h_{ND+f} \text{ -}\} \\ & fmap\, (fmap\, fst)\, (h_{State1}\, (Op\, (fmap\, h_{ND+f}\, (Inl\, (Get\, ((\Leftrightarrow) \circ k)))))) \\ = \;& \{\text{- definition of } fmap \text{ -}\} \\ & fmap\, (fmap\, fst)\, (h_{State1}\, (Op\, (Inl\, (Get\, (h_{ND+f} \circ (\Leftrightarrow) \circ k))))) \\ = \;& \{\text{- definition of } h_{State1} \text{ -}\} \\ & fmap\, (fmap\, fst)\, (\lambda s \to (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ k)\, s\, s) \\ = \;& \{\text{- definition of } fmap \text{ -}\} \\ & (\lambda s \to fmap\, fst\, ((h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ k)\, s\, s)) \\ = \;& \{\text{- definition of } fmap \text{ -}\} \\ & (\lambda s \to ((fmap\, (fmap\, fst) \circ h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ k)\, s\, s)) \\ = \;& \{\text{- define } alg^{S}_{LHS}\, (Get\, k) = \lambda s \to k\, s\, s \text{ -}\} \\ & alg^{S}_{LHS}\, (Get\, (h_{Global} \circ k)) \\ = \;& \{\text{- definition of } fmap \text{ -}\} \\ & alg^{S}_{LHS}\, (fmap\, h_{Global}\, (Get\, k)) \end{aligned}$$

case $op = Put\, s\, k$

$$\begin{aligned} & h_{Global}\, (alg\, (Inl\, (Put\, s\, k))) \\ = \;& \{\text{- definition of } alg \text{ -}\} \\ & h_{Global}\, (put_R\, s \gg k) \\ = \;& \{\text{- definition of } put_R \text{ -}\} \\ & h_{Global}\, ((get \gg\!= \lambda t \to put\, s \,[]\, side\, (put\, t)) \gg k) \\ = \;& \{\text{- definitions of } side, get, put, ([]), (\gg\!=) \text{ -}\} \\ & h_{Global}\, (Op\, (Inl\, (Get\, (\lambda t \to Op\, (Inr\, (Inl\, (Or\, (Op\, (Inl\, (Put\, s\, k))) \\ & \qquad\qquad (Op\, (Inl\, (Put\, t\, (Op\, (Inr\, (Inl\, Fail)))))))))))))) \end{aligned}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } h_{Global} \text{ -}\} \\
&\quad fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow) \\
&\qquad (Op\ (Inl\ (Get\ (\lambda t \to Op\ (Inr\ (Inl\ (Or\ (Op\ (Inl\ (Put\ s\ k))) \\
&\qquad\qquad\qquad\qquad (Op\ (Inl\ (Put\ t\ (Op\ Inr\ ((Inl\ Fail)))))))))))))))))) \\
&= \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
&\quad fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ (\\
&\qquad (Op\ (Inr\ (Inl\ (Get\ (\lambda t \to Op\ (Inl\ (Or\ (Op\ (Inr\ (Inl\ (Put\ s\ ((\Leftrightarrow)\ k))))) \\
&\qquad\qquad\qquad\qquad (Op\ (Inr\ (Inl\ (Put\ t\ (Op\ (Inl\ Fail)))))))))))))))))) \\
&= \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
&\quad fmap\ (fmap\ fst)\ (h_{State1}\ (\\
&\qquad (Op\ (Inl\ (Get\ (\lambda t \to liftM2\ (+\!\!+)\ (Op\ (Inl\ (Put\ s\ (h_{ND+f}\ ((\Leftrightarrow)\ k))))) \\
&\qquad\qquad\qquad\qquad (Op\ (Inl\ (Put\ t\ (Var\ [\,])))))))))) \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad fmap\ (fmap\ fst) \\
&\qquad (\lambda t \to h_{State1}\ (liftM2\ (+\!\!+)\ (Op\ (Inl\ (Put\ s\ (h_{ND+f}\ ((\Leftrightarrow)\ k))))) \\
&\qquad\qquad\qquad\qquad (Op\ (Inl\ (Put\ t\ (Var\ [\,])))))\ t) \\
&= \quad \{\text{- definition of } liftM2 \text{ -}\} \\
&\quad fmap\ (fmap\ fst) \\
&\qquad (\lambda t \to h_{State1}\ (\mathbf{do}\ x \leftarrow Op\ (Inl\ (Put\ s\ (h_{ND+f}\ ((\Leftrightarrow)\ k)))) \\
&\qquad\quad y \leftarrow Op\ (Inl\ (Put\ t\ (Var\ [\,]))) \\
&\qquad\quad Var\ (x +\!\!+ y) \\
&\qquad\quad)\ t) \\
&= \quad \{\text{- Lemma 4 -}\} \\
&\quad fmap\ (fmap\ fst) \\
&\qquad (\lambda t \to \mathbf{do}\ (x, t_1) \leftarrow h_{State1}\ (Op\ (Inl\ (Put\ s\ (h_{ND+f}\ ((\Leftrightarrow)\ k))))))\ t \\
&\qquad\quad (y, t_2) \leftarrow h_{State1}\ (Op\ (Inl\ (Put\ t\ (Var\ [\,]))))\ t_1 \\
&\qquad\quad h_{State1}\ (Var\ (x +\!\!+ y))\ t_2 \\
&\qquad)\ \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad fmap\ (fmap\ fst) \\
&\qquad (\lambda t \to \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ k))\ s \\
&\qquad\quad (y, t_2) \leftarrow Var\ ([\,], t) \\
&\qquad\quad Var\ (x +\!\!+ y, t_2) \\
&\qquad)\ \\
&= \quad \{\text{- monad law -}\} \\
&\quad fmap\ (fmap\ fst) \\
&\qquad (\lambda t \to \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ k))\ s \\
&\qquad\quad Var\ (x +\!\!+ [\,], t) \\
&\qquad)\ \\
&= \quad \{\text{- right unit of } (+\!\!+) \text{ -}\} \\
&\quad fmap\ (fmap\ fst) \\
&\qquad (\lambda t \to \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ k))\ s \\
&\qquad\quad Var\ (x, t) \\
&\qquad)\ \\
&= \quad \{\text{- definition of } fmap\ fst \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & fmap\ (fmap\ fst) \\
 & \quad (\lambda t \to \mathbf{do}\ x \leftarrow fmap\ fst\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ k))\ s) \\
 & \quad\quad Var\ (x, t) \\
 & \quad) \\
= & \ \{\text{- definition of } fmap \text{ -}\} \\
 & fmap\ (fmap\ fst) \\
 & \quad (\lambda t \to \mathbf{do}\ x \leftarrow (fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ k))))\ s \\
 & \quad\quad Var\ (x, t) \\
 & \quad) \\
= & \ \{\text{- definition of } fmap\ (fmap\ fst) \text{ -}\} \\
 & \backslash\_ \to \mathbf{do}\ x \leftarrow (fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ k))))\ s \\
 & \quad Var\ x \\
= & \ \{\text{- monad law -}\} \\
 & \backslash\_ \to (fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ k))))\ s \\
= & \ \{\text{- definition of } h_{Global} \text{ -}\} \\
 & \backslash\_ \to (h_{Global}\ k)\ s \\
= & \ \{\text{- define } alg^{S}_{LHS}\ (Put\ s\ k) = \backslash\_ \to k\ s \text{ -}\} \\
 & alg^{S}_{LHS}\ (Put\ s\ (h_{Global}\ k)) \\
= & \ \{\text{- definition of } fmap \text{ -}\} \\
 & alg^{S}_{LHS}\ (fmap\ h_{Global}\ (Put\ s))
\end{array}
$$

We conclude that this fusion subcondition holds provided that:

$$
\begin{array}{l}
alg^{S}_{LHS} :: Functor\ f \Rightarrow State_F\ s\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a]) \\
alg^{S}_{LHS}\ (Get\ k)\ \ = \lambda s \to k\ s\ s \\
alg^{S}_{LHS}\ (Put\ s\ k) = \backslash\_ \to k\ s
\end{array}
$$

Let's consider the second subcondition. It has also two cases:
case $op = Fail$

$$
\begin{array}{ll}
 & h_{Global}\ (alg\ (Inr\ (Inl\ Fail))) \\
= & \ \{\text{- definition of } alg \text{ -}\} \\
 & h_{Global}\ (Op\ (Inr\ (Inl\ Fail))) \\
= & \ \{\text{- definition of } h_{Global} \text{ -}\} \\
 & fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inl\ Fail))))))) \\
= & \ \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
 & fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ (Op\ (Inl\ (fmap\ (\Leftrightarrow)\ Fail))))) \\
= & \ \{\text{- definition of } fmap \text{ -}\} \\
 & fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ (Op\ (Inl\ Fail)))) \\
= & \ \{\text{- definition of } h_{ND+f} \text{ -}\} \\
 & fmap\ (fmap\ fst)\ (h_{State1}\ (Var\ [\,])) \\
= & \ \{\text{- definition of } h_{State1} \text{ -}\} \\
 & fmap\ (fmap\ fst)\ (\lambda s \to Var\ ([\,], s)) \\
= & \ \{\text{- definition of } fmap \text{ twice and } fst \text{ -}\} \\
 & \lambda s \to Var\ [\,] \\
= & \ \{\text{- define } alg^{ND}_{RHS}\ Fail = \lambda s \to Var\ [\,] \text{ -}\} \\
 & alg^{ND}_{RHS}\ Fail
\end{array}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
&\quad \mathit{alg}^{ND}_{RHS}\ (\mathit{fmap}\ h_{Global}\ \mathit{Fail})
\end{aligned}
$$

case $op = Or\ p\ q$

$$
\begin{aligned}
&\quad h_{Global}\ (\mathit{alg}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Or}\ p\ q)))) \\
&= \quad \{\text{- definition of } \mathit{alg} \text{ -}\} \\
&\quad h_{Global}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Or}\ p\ q)))) \\
&= \quad \{\text{- definition of } h_{Global} \text{ -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Or}\ p\ q))))))) \\
&= \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{State1}\ (h_{ND+f}\ (\mathit{Op}\ (\mathit{Inl}\ (\mathit{fmap}\ (\Leftrightarrow)\ (\mathit{Or}\ p\ q))))))) \\
&= \quad \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{State1}\ (h_{ND+f}\ (\mathit{Op}\ (\mathit{Inl}\ (\mathit{Or}\ ((\Leftrightarrow)\ p)\ ((\Leftrightarrow)\ q))))))) \\
&= \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{State1}\ (\mathit{liftM2}\ (+\!\!+)\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ (h_{ND+f}\ ((\Leftrightarrow)\ q)))) \\
&= \quad \{\text{- definition of } \mathit{liftM2} \text{ -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{State1}\ (\mathbf{do}\ x \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ p) \\
&\qquad\qquad\qquad\qquad\qquad\quad y \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ q) \\
&\qquad\qquad\qquad\qquad\qquad\quad \eta\ (x +\!\!+ y))) \\
&= \quad \{\text{- Lemma 4 -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s_0 \rightarrow (\mathbf{do}\ (x, s_1) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_0 \\
&\qquad\qquad\qquad\qquad\qquad\quad (y, s_2) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s_1 \\
&\qquad\qquad\qquad\qquad\qquad\quad h_{State1}\ (\eta\ (x +\!\!+ y))\ s_2)) \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s_0 \rightarrow (\mathbf{do}\ (x, s_1) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_0 \\
&\qquad\qquad\qquad\qquad\qquad\quad (y, s_2) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s_1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Var}\ (x +\!\!+ y, s_2))) \\
&= \quad \{\text{- Lemma 1 } (p \text{ and } q \text{ are in the codomain of } \mathit{local2global}) \text{ -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s_0 \rightarrow (\mathbf{do}\ (x, s_1) \leftarrow \mathbf{do}\ \{(x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_0; \eta\ (x, s_0)\} \\
&\qquad\qquad\qquad\qquad\qquad\quad (y, s_2) \leftarrow \mathbf{do}\ \{(y, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s_1; \eta\ (x, s_1)\} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Var}\ (x +\!\!+ y, s_2))) \\
&= \quad \{\text{- monad laws -}\} \\
&\quad \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s_0 \rightarrow (\mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_0 \\
&\qquad\qquad\qquad\qquad\qquad\quad (y, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s_0 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Var}\ (x +\!\!+ y, s_0))) \\
&= \quad \{\text{- definition of } \mathit{fmap} \text{ (twice) and } \mathit{fst} \text{ -}\} \\
&\quad \lambda s_0 \rightarrow (\mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_0 \\
&\qquad\qquad\qquad\quad (y, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s_0 \\
&\qquad\qquad\qquad\quad \mathit{Var}\ (x +\!\!+ y)) \\
&= \quad \{\text{- definition of } \mathit{fmap}, \mathit{fst} \text{ and monad laws -}\} \\
&\quad \lambda s_0 \rightarrow (\mathbf{do}\ x \leftarrow \mathit{fmap}\ \mathit{fst}\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_0) \\
&\qquad\qquad\qquad\quad y \leftarrow \mathit{fmap}\ \mathit{fst}\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s_0) \\
&\qquad\qquad\qquad\quad \mathit{Var}\ (x +\!\!+ y)) \\
&= \quad \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
&\quad \lambda s_0 \rightarrow (\mathbf{do}\ x \leftarrow \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p)))\ s_0
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&\quad\ y \leftarrow fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q)))\ s_0 \\
&\quad\ Var\ (x +\!\!+ y)) \\
&= \ \{\text{- definition of } h_{Global} \text{ -}\} \\
&\ \ \lambda s_0 \rightarrow (\mathbf{do}\ x \leftarrow h_{Global}\ p\ s_0 \\
&\quad\ y \leftarrow h_{Global}\ q\ s_0 \\
&\quad\ Var\ (x +\!\!+ y)) \\
&= \ \{\text{- definition of } liftM2 \text{ -}\} \\
&\ \ \lambda s_0 \rightarrow liftM2\ (+\!\!+)\ (h_{Global}\ p\ s_0)\ (h_{Global}\ q\ s_0) \\
&= \ \{\text{- define } alg^{ND}_{LHS}\ (Or\ p\ q) = \lambda s \rightarrow liftM2\ (+\!\!+)\ (p\ s)\ (q\ s) \text{ -}\} \\
&\ \ alg^{ND}_{LHS}\ (Or\ (h_{Global}\ p)\ (h_{Global}\ q)) \\
&= \ \{\text{- definition of } fmap \text{ -}\} \\
&\ \ alg^{ND}_{LHS}\ (fmap\ hGobal\ (Or\ p\ q))
\end{aligned}
$$

We conclude that this fusion subcondition holds provided that:

$$
\begin{aligned}
&alg^{ND}_{LHS} :: Functor\ f \Rightarrow Nondet_F\ (s \rightarrow Free\ f\ [a]) \rightarrow (s \rightarrow Free\ f\ [a]) \\
&alg^{ND}_{LHS}\ Fail \quad\ = \lambda s \rightarrow Var\ [\,] \\
&alg^{ND}_{LHS}\ (Or\ p\ q) = \lambda s \rightarrow liftM2\ (+\!\!+)\ (p\ s)\ (q\ s)
\end{aligned}
$$

Finally, the last subcondition:

$$
\begin{aligned}
&\quad\ h_{Global}\ (alg\ (Inr\ (Inr\ op))) \\
&= \ \{\text{- definition of } alg \text{ -}\} \\
&\ \ h_{Global}\ (Op\ (Inr\ (Inr\ op))) \\
&= \ \{\text{- definition of } h_{Global} \text{ -}\} \\
&\ \ fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inr\ op))))))) \\
&= \ \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
&\ \ fmap\ (fmap\ fst)\ (h_{State1}\ (h_{ND+f}\ (Op\ (Inr\ (Inr\ (fmap\ (\Leftrightarrow)\ op))))))) \\
&= \ \{\text{- definition of } h_{ND+f} \text{ -}\} \\
&\ \ fmap\ (fmap\ fst)\ (h_{State1}\ (Op\ (fmap\ h_{ND+f}\ (Inr\ (fmap\ (\Leftrightarrow)\ op))))) \\
&= \ \{\text{- definition of } fmap \text{ -}\} \\
&\ \ fmap\ (fmap\ fst)\ (h_{State1}\ (Op\ (Inr\ (fmap\ h_{ND+f}\ (fmap\ (\Leftrightarrow)\ op))))) \\
&= \ \{\text{- } fmap \text{ fusion -}\} \\
&\ \ fmap\ (fmap\ fst)\ (h_{State1}\ (Op\ (Inr\ (fmap\ (h_{ND+f} \circ (\Leftrightarrow))\ op)))) \\
&= \ \{\text{- definition of } h_{State1} \text{ -}\} \\
&\ \ fmap\ (fmap\ fst)\ (\lambda s \rightarrow Op\ (fmap\ (\$s)\ (fmap\ h_{State1}\ (fmap\ (h_{ND+f} \circ (\Leftrightarrow))\ op)))) \\
&= \ \{\text{- } fmap \text{ fusion -}\} \\
&\ \ fmap\ (fmap\ fst)\ (\lambda s \rightarrow Op\ (fmap\ (\$s)\ (fmap\ (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow))\ op))) \\
&= \ \{\text{- definition of } fmap \text{ -}\} \\
&\ \ \lambda s \rightarrow fmap\ fst\ (Op\ (fmap\ (\$s)\ (fmap\ (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow))\ op))) \\
&= \ \{\text{- definition of } fmap \text{ -}\} \\
&\ \ \lambda s \rightarrow Op\ (fmap\ (fmap\ fst)\ (fmap\ (\$s)\ (fmap\ (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow))\ op))) \\
&= \ \{\text{- } fmap \text{ fusion -}\} \\
&\ \ \lambda s \rightarrow Op\ (fmap\ (fmap\ fst \circ (\$s))\ (fmap\ (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow))\ op))) \\
&= \ \{\text{- Lemma 2 -}\} \\
&\ \ \lambda s \rightarrow Op\ (fmap\ ((\$s) \circ fmap\ (fmap\ fst))\ (fmap\ (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow))\ op))) \\
&= \ \{\text{- } fmap \text{ fission -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \lambda s \to Op\ ((fmap\ (\$s) \circ fmap\ (fmap\ (fmap\ fst)))\ (fmap\ (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow))\ op)) \\
= \quad & \{\text{- } fmap \text{ fusion -}\} \\
& \lambda s \to Op\ (fmap\ (\$s)\ (fmap\ (fmap\ (fmap\ fst) \circ h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow))\ op)) \\
= \quad & \{\text{- definition of } h_{Global} \text{ -}\} \\
& \lambda s \to Op\ (fmap\ (\$s)\ (fmap\ h_{Global}\ op)) \\
= \quad & \{\text{- define } fwd_{LHS}\ op = \lambda s \to Op\ (fmap\ (\$s)\ op) \text{ -}\} \\
& fwd_{LHS}\ (fmap\ h_{Global}\ op)
\end{aligned}
$$

We conclude that this fusion subcondition holds provided that:

$$
\begin{aligned}
& fwd_{LHS} :: Functor\ f \Rightarrow f\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a]) \\
& fwd_{LHS}\ op = \lambda s \to Op\ (fmap\ (\$s)\ op)
\end{aligned}
$$

2.4 Equating the fused sides

We observe that the following equations hold trivially.

$$
\begin{aligned}
gen_{LHS} &= gen_{RHS} \\
alg^{S}_{LHS} &= alg^{S}_{RHS} \\
alg^{ND}_{LHS} &= alg^{ND}_{RHS} \\
fwd_{LHS} &= fwd_{RHS}
\end{aligned}
$$

Therefore, the main theorem holds.

2.5 Key lemma: State restoration

The key lemma is the following, which guarantees that *local2global* restores the initial state after a computation.

Lemma 1 (State is Restored)**.**

$$
\begin{gathered}
h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ t)))\ s \\
= \\
\mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ t)))\ s;\ \eta\ (x, s)
\end{gathered}
$$

Proof The proof proceeds by structural induction on t.

case $t = Var\ y$

$$
\begin{aligned}
& h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Var\ y))))\ s \\
= \quad & \{\text{- definition of } local2global \text{ -}\} \\
& h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Var\ y)))\ s \\
= \quad & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& h_{State1}\ (h_{ND+f}\ (Var\ y))\ s \\
= \quad & \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& h_{State1}\ (Var\ [y])\ s
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad\ Var\ ([y], s) \\
&= \quad \{\text{- monad law -}\} \\
&\quad\ \mathbf{do}\ (x, \_) \leftarrow Var\ ([y], s);\ Var\ (x, s) \\
&= \quad \{\text{- definition of } local2global, h_{ND+f}, (\Leftrightarrow), h_{State1} \text{ and } \eta \text{ -}\} \\
&\quad\ \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Var\ y))))\ s;\ \eta\ (x, s)
\end{aligned}
$$

case $t = Op\ (Inl\ (Get\ k))$

$$
\begin{aligned}
&\quad\ h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Op\ (Inl\ (Get\ k))))))\ s \\
&= \quad \{\text{- definition of } local2global \text{ -}\} \\
&\quad\ h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inl\ (Get\ (local2global \circ k))))))\ s \\
&= \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
&\quad\ h_{State1}\ (h_{ND+f}\ (Op\ (Inr\ (Inl\ (Get\ ((\Leftrightarrow) \circ local2global \circ k))))))\ s \\
&= \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
&\quad\ h_{State1}\ (Op\ (Inl\ (Get\ (h_{ND+f} \circ (\Leftrightarrow) \circ local2global \circ k))))\ s \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad\ (h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global \circ k)\ s\ s \\
&= \quad \{\text{- definition of } (\circ) \text{ -}\} \\
&\quad\ (h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (k\ s)))))\ s \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad\ \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ ((\Leftrightarrow)\ (h_{ND+f}\ (local2global\ (k\ s))))\ s;\ \eta\ (x, s) \\
&= \quad \{\text{- definition of } local2global, (\Leftrightarrow), h_{ND+f}, h_{State1} \text{ -}\} \\
&\quad\ \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ (local2global\ (Op\ (Inl\ (Get\ k)))))\ s;\ \eta\ (x, s)
\end{aligned}
$$

case $t = Op\ (Inr\ (Inl\ Fail))$

$$
\begin{aligned}
&\quad\ h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Op\ (Inr\ (Inl\ Fail))))))\ s \\
&= \quad \{\text{- definition of } local2global \text{ -}\} \\
&\quad\ h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inl\ Fail)))))\ s \\
&= \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
&\quad\ h_{State1}\ (h_{ND+f}\ (Op\ (Inl\ Fail)))\ s \\
&= \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
&\quad\ h_{State1}\ (Var\ [\,])\ s \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad\ Var\ ([\,], s) \\
&= \quad \{\text{- monad law -}\} \\
&\quad\ \mathbf{do}\ (x, \_) \leftarrow Var\ ([\,], s);\ Var\ (x, s) \\
&= \quad \{\text{- definition of } local2global, (\Leftrightarrow), h_{ND+f}, h_{State1} \text{ -}\} \\
&\quad\ \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Op\ (Inr\ (Inl\ Fail))))))\ s;\ \eta\ (x, s)
\end{aligned}
$$

case $t = Op\ (Inl\ (Put\ t\ k))$

$$
\begin{aligned}
&\quad\ h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Op\ (Inl\ (Put\ t\ k))))))\ s \\
&= \quad \{\text{- definition of } local2global \text{ -}\} \\
&\quad\ h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (put_R\ t \gg local2global\ k)))\ s \\
&= \quad \{\text{- definition of } put_R \text{ -}\} \\
&\quad\ h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ ((get \gg\!= \lambda t' \rightarrow put\ t\ [\!]\ side\ (put\ t')) \gg local2global\ k)))\ s
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } (\llbracket\rrbracket),\ \mathit{get},\ \mathit{put},\ \mathit{side} \text{ and } (\gg\!=) \text{ -}\}\\
&\quad h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inl\ (Get\ (\lambda t' \rightarrow\\
&\qquad\qquad Op\ (Inr\ (Inl\ (Or\ (Op\ (Inl\ (Put\ t\ (local2global\ k))))\\
&\qquad\qquad\qquad\qquad (Op\ (Inl\ (Put\ t'\ (Op\ (Inr\ (Inl\ Fail))))))))))))))\ s\\
&= \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\}\\
&\quad h_{State1}\ (h_{ND+f}\ (Op\ (Inr\ (Inl\ (Get\ (\lambda t' \rightarrow\\
&\qquad Op\ (Inl\ (Or\ (Op\ (Inr\ (Inl\ (Put\ t\ ((\Leftrightarrow)\ (local2global\ k))))))\\
&\qquad\qquad (Op\ (Inr\ (Inl\ (Put\ t'\ (Op\ (Inl\ Fail))))))))))))\ s\\
&= \quad \{\text{- definition of } h_{ND+f} \text{ -}\}\\
&\quad h_{State1}\ (Op\ (Inl\ (Get\ (\lambda t' \rightarrow\\
&\qquad liftM2\ (+\!\!+)\ (Op\ (Inl\ (Put\ t\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k))))))\\
&\qquad\qquad (Op\ (Inl\ (Put\ t'\ (Var\ [\,]))))))))\ s\\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\}\\
&\quad h_{State1}\ (liftM2\ (+\!\!+)\ (Op\ (Inl\ (Put\ t\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k))))))\\
&\qquad\qquad (Op\ (Inl\ (Put\ s\ (Var\ [\,]))))))\ s\\
&= \quad \{\text{- definition of } liftM2 \text{ -}\}\\
&\quad h_{State1}\ (\mathbf{do}\ x \leftarrow Op\ (Inl\ (Put\ t\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k))))))\\
&\qquad\qquad y \leftarrow Op\ (Inl\ (Put\ s\ (Var\ [\,])))\\
&\qquad\qquad Var\ (x +\!\!+ y)\\
&\qquad)\ s\\
&= \quad \{\text{- Lemma 4 -}\}\\
&\quad \mathbf{do}\ (x, s_1) \leftarrow h_{State1}\ (Op\ (Inl\ (Put\ t\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k))))))\ s\\
&\qquad (y, s_2) \leftarrow h_{State1}\ (Op\ (Inl\ (Put\ s\ (Var\ [\,]))))\ s_1\\
&\qquad Var\ (x +\!\!+ y, s_2)\\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\}\\
&\quad \mathbf{do}\ (x, s_1) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k)))\ t\\
&\qquad (y, s_2) \leftarrow Var\ ([\,], s)\\
&\qquad Var\ (x +\!\!+ y, s_2)\\
&= \quad \{\text{- monad laws -}\}\\
&\quad \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k)))\ t\\
&\qquad Var\ (x +\!\!+ [\,], s)\\
&= \quad \{\text{- right unit of } (+\!\!+) \text{ -}\}\\
&\quad \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k)))\ t\\
&\qquad Var\ (x, s)\\
&= \quad \{\text{- monad laws -}\}\\
&\quad \mathbf{do}\ (x, \_) \leftarrow \mathbf{do}\ \{(x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ k)))\ t;\ \eta\ (x, s)\}\\
&\qquad Var\ (x, s)\\
&= \quad \{\text{- deriviation in reverse -}\}\\
&\quad \mathbf{do}\ (x, \_) \leftarrow h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Op\ (Inl\ (Put\ t\ k))))))\ s\\
&\qquad Var\ (x, s)
\end{aligned}
$$

case $t = Op\ (Inr\ (Inl\ (Or\ p\ q)))$

$$
\begin{aligned}
&\quad h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global\ (Op\ (Inr\ (Inl\ (Or\ p\ q)))))))\ s\\
&= \quad \{\text{- definition of } local2global \text{ -}\}\\
&\quad h_{State1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inl\ (Or\ (local2global\ p)\ (local2global\ q)))))))\ s\\
&= \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_{State1}\,(h_{ND+f}\,(Op\,(Inl\,(Or\,((\Leftrightarrow)\,(local2global\,p))\,((\Leftrightarrow)\,(local2global\,q))))))\,s \\
= \;& \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& h_{State1}\,(liftM2\,(+\!\!+)\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,p)))\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,q))))\,s \\
= \;& \{\text{- definition of } liftM2 \text{ -}\} \\
& h_{State1}\,(\mathbf{do}\;x \leftarrow h_{ND+f}\,((\Leftrightarrow)\,(local2global\,p)) \\
& \qquad\qquad y \leftarrow h_{ND+f}\,((\Leftrightarrow)\,(local2global\,q)) \\
& \qquad\qquad Var\,(x +\!\!+ y) \\
& \quad)\,s \\
= \;& \{\text{- Lemma 4 -}\} \\
& \mathbf{do}\;(x, s_1) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,p)))\,s \\
& \quad (y, s_2) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,q)))\,s_1 \\
& \quad h_{State1}\,(Var\,(x +\!\!+ y))\,s_2 \\
= \;& \{\text{- induction hypothesis -}\} \\
& \mathbf{do}\;(x, s_1) \leftarrow \mathbf{do}\;\{(x, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,p)))\,s;\,\eta\,(x, s)\} \\
& \quad (y, s_2) \leftarrow \mathbf{do}\;\{(y, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,q)))\,s_1;\,\eta\,(y, s_1)\} \\
& \quad h_{State1}\,(Var\,(x +\!\!+ y))\,s_2 \\
= \;& \{\text{- monad laws -}\} \\
& \mathbf{do}\;(x, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,p)))\,s \\
& \quad (y, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,q)))\,s_1 \\
& \quad h_{State1}\,(Var\,(x +\!\!+ y))\,s \\
= \;& \{\text{- definition of } h_{State1} \text{ -}\} \\
& \mathbf{do}\;(x, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,p)))\,s \\
& \quad (y, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,q)))\,s \\
& \quad \eta\,(x +\!\!+ y, s) \\
= \;& \{\text{- monad laws -}\} \\
& \mathbf{do}\;(x, \_) \leftarrow (\\
& \qquad \mathbf{do}\;(x, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,p)))\,s \\
& \qquad\quad (y, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,q)))\,s_1 \\
& \qquad\quad \eta\,(x +\!\!+ y, s) \\
& \qquad) \\
& \quad \eta\,(x, s) \\
= \;& \{\text{- derivation in reverse (similar to before) -}\} \\
& \mathbf{do}\;(x, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,(Op\,(Inr\,(Inl\,(Or\,p\,q))))))\,s \\
& \quad \eta\,(x, s)
\end{aligned}
$$

case $t = Op\,(Inr\,(Inr\,y))$

$$
\begin{aligned}
& h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,(Op\,(Inr\,(Inr\,y))))))\,s \\
= \;& \{\text{- definition of } local2global \text{ -}\} \\
& h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(Op\,(Inr\,(Inr\,(fmap\;local2global\,y))))))\,s \\
= \;& \{\text{- definition of } (\Leftrightarrow);\, fmap \text{ fusion -}\} \\
& h_{State1}\,(h_{ND+f}\,(Op\,(Inr\,(Inr\,(fmap\,((\Leftrightarrow) \circ local2global)\,y)))))\,s \\
= \;& \{\text{- definition of } h_{ND+f};\, fmap \text{ fusion -}\} \\
& h_{State1}\,(Op\,(Inr\,(fmap\,(h_{ND+f} \circ (\Leftrightarrow) \circ local2global)\,y)))\,s \\
= \;& \{\text{- definition of } h_{State1};\, fmap \text{ fusion -}\} \\
& Op\,(fmap\,((\$s) \circ h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global)\,y)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- induction hypothesis -}\}\\
&\quad Op\,(fmap\,((\gg\!= \lambda(x, \_) \to \eta\,(x, s)) \circ (\$s) \circ h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global)\,y)\\
&= \quad \{\text{- } fmap \text{ fission; definition of } (\gg\!=) \text{ -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow Op\,(fmap\,((\$s) \circ h_{State1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global)\,y)\\
&\qquad \eta\,(x, s)\\
&= \quad \{\text{- deriviation in reverse (similar to before) -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow h_{State1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global\,(Op\,(Inr\,(Inr\,y))))))\,s\\
&\qquad \eta\,(x, s)
\end{aligned}
$$

■

2.6 Auxiliary lemmas

The derivations above make use of two auxiliary lemmas. We prove them here.

Lemma 2 (Naturality of ($s)). $(\$x) \circ fmap\,f = f \circ (\$x)$

Proof

$$
\begin{aligned}
&\quad ((\$x) \circ fmap\,f)\,m\\
&= \quad \{\text{- function application -}\}\\
&\quad (fmap\,f\ m)\,x\\
&= \quad \{\text{- eta-expansion -}\}\\
&\quad (fmap\,f\ (\lambda y \circ m\,y))\,x\\
&= \quad \{\text{- definition of } fmap \text{ -}\}\\
&\quad (\lambda y \circ f\ (m\,y))\,x\\
&= \quad \{\text{- function application -}\}\\
&\quad f\,(m\,x)\\
&= \quad \{\text{- definition of } \circ \text{ and } \$ \text{ -}\}\\
&\quad (f \circ (\$x))\,m
\end{aligned}
$$

■

Lemma 3.

$$fmap\,(fmap\,fst)\,(liftM2\,(+\!+)\,p\,q) = liftM2\,(+\!+)\,(fmap\,(fmap\,fst)\,p)\,(fmap\,(fmap\,fst)\,q)$$

Proof

$$
\begin{aligned}
&\quad fmap\,(fmap\,fst)\,(liftM2\,(+\!+)\,p\,q)\\
&= \quad \{\text{- definition of } liftM2 \text{ -}\}\\
&\quad fmap\,(fmap\,fst)\,(\mathbf{do}\,\{x \leftarrow p; y \leftarrow q; \eta\,(x +\!+ y)\})\\
&= \quad \{\text{- derived property for monad: } fmap\,f\,(m \gg\!= k) = m \gg\!= fmap\,f \circ k \text{ -}\}\\
&\quad \mathbf{do}\,\{x \leftarrow p; y \leftarrow q; fmap\,(fmap\,fst)\,(\eta\,(x +\!+ y))\}\\
&= \quad \{\text{- definition of } fmap \text{ -}\}\\
&\quad \mathbf{do}\,\{x \leftarrow p; y \leftarrow q; \eta\,(fmap\,fst\,(x +\!+ y))\}\\
&= \quad \{\text{- naturality of } (+\!+) \text{ -}\}\\
&\quad \mathbf{do}\,\{x \leftarrow p; y \leftarrow q; \eta\,((fmap\,fst\,x) +\!+ (fmap\,fst\,y))\}\\
&= \quad \{\text{- monad left unit law (twice) -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathbf{do}\,\{x \leftarrow p; x' \leftarrow \eta\,(\mathit{fmap}\,\mathit{fst}\,x); y \leftarrow q; y' \leftarrow \eta\,(\mathit{fmap}\,\mathit{fst}\,y)\,\eta\,(x' +\!\!+ y')\} \\
= \;& \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
& \mathbf{do}\,\{x \leftarrow \mathit{fmap}\,(\mathit{fmap}\,\mathit{fst})\,p; y \leftarrow \mathit{fmap}\,(\mathit{fmap}\,\mathit{fst})\,q; \eta\,(x +\!\!+ y)\} \\
= \;& \{\text{- definition of } \mathit{liftM2} \text{ -}\} \\
& \mathit{liftM2}\,(+\!\!+)\,(\mathit{fmap}\,(\mathit{fmap}\,\mathit{fst})\,p)\,(\mathit{fmap}\,(\mathit{fmap}\,\mathit{fst})\,q)
\end{aligned}
$$

■

Lemma 4 (Distributivity of h_{State1})**.**

$$h_{State1}\,(p \gg\!= k)\,s = h_{State1}\,p\,s \gg\!= \lambda(x, s') \rightarrow h_{State1}\,(k\,x)\,s'$$

Proof The proof proceeds by induction on p.

case $p = \mathit{Var}\,x$

$$
\begin{aligned}
& h_{State1}\,(\mathit{Var}\,x \gg\!= k)\,s \\
= \;& \{\text{- monad law -}\} \\
& h_{State1}\,(k\,x)\,s \\
= \;& \{\text{- monad law -}\} \\
& \eta\,(x, s) \gg\!= \lambda(x, s') \rightarrow h_{State1}\,(k\,x)\,s' \\
= \;& \{\text{- definition of } h_{State1} \text{ -}\} \\
& h_{State1}\,(\mathit{Var}\,x)\,s \gg\!= \lambda(x, s') \rightarrow h_{State1}\,(k\,x)\,s'
\end{aligned}
$$

case $p = \mathit{Op}\,(\mathit{Inl}\,(\mathit{Get}\,p))$

$$
\begin{aligned}
& h_{State1}\,(\mathit{Op}\,(\mathit{Inl}\,(\mathit{Get}\,p)) \gg\!= k)\,s \\
= \;& \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
& h_{State1}\,(\mathit{Op}\,(\mathit{fmap}\,(\gg\!= k)\,(\mathit{Inl}\,(\mathit{Get}\,p))))\,s \\
= \;& \{\text{- definition of } \mathit{fmap} \text{ for coproduct } (:+:) \text{ -}\} \\
& h_{State1}\,(\mathit{Op}\,(\mathit{Inl}\,(\mathit{fmap}\,(\gg\!= k)\,(\mathit{Get}\,p))))\,s \\
= \;& \{\text{- definition of } \mathit{fmap} \text{ for } \mathit{Get} \text{ -}\} \\
& h_{State1}\,(\mathit{Op}\,(\mathit{Inl}\,(\mathit{Get}\,(\lambda x \rightarrow p\,s \gg\!= k))))\,s \\
= \;& \{\text{- definition of } h_{State1} \text{ -}\} \\
& h_{State1}\,(p\,s \gg\!= k)\,s \\
= \;& \{\text{- induction hypothesis -}\} \\
& h_{State1}\,(p\,s)\,s \gg\!= \lambda(x, s') \rightarrow h_{State1}\,(k\,x)\,s' \\
= \;& \{\text{- definition of } h_{State1} \text{ -}\} \\
& h_{State1}\,(\mathit{Op}\,(\mathit{Inl}\,(\mathit{Get}\,p)))\,s \gg\!= \lambda(x, s') \rightarrow h_{State1}\,(k\,x)\,s'
\end{aligned}
$$

case $p = \mathit{Op}\,(\mathit{Inl}\,(\mathit{Put}\,t\,p))$

$$
\begin{aligned}
& h_{State1}\,(\mathit{Op}\,(\mathit{Inl}\,(\mathit{Put}\,t\,p)) \gg\!= k)\,s \\
= \;& \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
& h_{State1}\,(\mathit{Op}\,(\mathit{fmap}\,(\gg\!= k)\,(\mathit{Inl}\,(\mathit{Put}\,t\,p))))\,s \\
= \;& \{\text{- definition of } \mathit{fmap} \text{ for coproduct } (:+:) \text{ -}\} \\
& h_{State1}\,(\mathit{Op}\,(\mathit{Inl}\,(\mathit{fmap}\,(\gg\!= k)\,(\mathit{Put}\,t\,p))))\,s \\
= \;& \{\text{- definition of } \mathit{fmap} \text{ for } \mathit{Put} \text{ -}\} \\
& h_{State1}\,(\mathit{Op}\,(\mathit{Inl}\,(\mathit{Put}\,t\,(p \gg\!= k))))\,s
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad h_{State1}\ (p \gg\!= k)\ t \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad h_{State1}\ p\ t \gg\!= \lambda(x, s') \to h_{State1}\ (k\ x)\ s' \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad h_{State1}\ (Op\ (Inl\ (Put\ t\ p)))\ s \gg\!= \lambda(x, s') \to h_{State1}\ (k\ x)\ s'
\end{aligned}
$$

case $p = Op\ (Inr\ y)$

$$
\begin{aligned}
&\quad h_{State1}\ (Op\ (Inr\ y) \gg\!= k)\ s \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
&\quad h_{State1}\ (Op\ (fmap\ (\gg\!= k)\ (Inr\ y)))\ s \\
&= \quad \{\text{- definition of } fmap \text{ for coproduct } (:+:) \text{ -}\} \\
&\quad h_{State1}\ (Op\ (Inr\ (fmap\ (\gg\!= k)\ y)))\ s \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad Op\ (fmap\ (\lambda x \to h_{State1}\ x\ s)\ (fmap\ (\gg\!= k)\ y)) \\
&= \quad \{\text{- } fmap \text{ fusion -}\} \\
&\quad Op\ (fmap\ ((\lambda x \to h_{State1}\ (x \gg\!= k)\ s))\ y) \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad Op\ (fmap\ (\lambda x \to h_{State1}\ x\ s \gg\!= \lambda(x', s') \to h_{State1}\ (k\ x')\ s')\ y) \\
&= \quad \{\text{- } fmap \text{ fission -}\} \\
&\quad Op\ (fmap\ (\lambda x \to x \gg\!= \lambda(x', s') \to h_{State1}\ (k\ x')\ s')\ (fmap\ (\lambda x \to h_{State1}\ x\ s)\ y)) \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ -}\} \\
&\quad Op\ ((fmap\ (\lambda x \to h_{State1}\ x\ s)\ y)) \gg\!= \lambda(x', s') \to h_{State1}\ (k\ x')\ s' \\
&= \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
&\quad Op\ (Inr\ y)\ s \gg\!= \lambda(x', s') \to h_{State1}\ (k\ x')\ s'
\end{aligned}
$$

■

```
import Data.Bitraversable (Bitraversable)
```

3 Proofs for modelling nondeterminism with state

In this section, we prove the theorems in Section 5.

3.1 Only nondeterminism

This section proves the following theorem in Section 5.1.

Theorem 2. $run_{ND} = h_{ND}$

Proof We start with expanding the definition of run_{ND}:

$$extract_S \circ h'_{State} \circ nondet2state_S = h_{ND}$$

Both $nondet2state_S$ and h_{ND} are written as folds. We use the fold fusion law **fusion-post'** (3.3) to fuse the left-hand side. Since the right-hand side is already a fold, to prove the equation we just need to check the components of the fold h_{ND} satisfy the conditions

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

of the fold fusion, i.e., the following two equations:

$$\begin{aligned} & (extract_S \circ h'_{State}) \circ gen = gen_{ND} \\ & (extract_S \circ h'_{State}) \circ alg \circ fmap\ nondet2state_S \\ = \quad & alg_{ND} \circ fmap\ (extract_S \circ h'_{State}) \circ fmap\ nondet2state_S \end{aligned}$$

For brevity, we omit the last common part $fmap\ nondet2state_S$ of the second equation in the following proof. Instead, we assume that the input is in the codomain of $fmap\ nondet2state_S$.

For the first equation, we calculate as follows:

$$\begin{aligned} & extract_S\ (h'_{State}\ (gen\ x)) \\ = \quad & \{\text{- definition of } gen \text{ -}\} \\ & extract_S\ (h'_{State}\ (append_S\ x\ pop_S)) \\ = \quad & \{\text{- definition of } extract_S \text{ -}\} \\ & results \circ snd\ \$\ run_{State}\ (h'_{State}\ (append_S\ x\ pop_S))\ (S\ [\,]\ [\,]) \\ = \quad & \{\text{- Lemma 6 -}\} \\ & results \circ snd\ \$\ run_{State}\ (h'_{State}\ pop_S)\ (S\ ([\,] +\!\!+ [x])\ [\,]) \\ = \quad & \{\text{- definition of } (+\!\!+) \text{ -}\} \\ & results \circ snd\ \$\ run_{State}\ (h'_{State}\ pop_S)\ (S\ [x]\ [\,]) \\ = \quad & \{\text{- Lemma 7 -}\} \\ & results \circ snd\ \$\ ((), S\ [x]\ [\,]) \\ = \quad & \{\text{- definition of } snd \text{ -}\} \\ & results\ (S\ [x]\ [\,]) \\ = \quad & \{\text{- definition of } results \text{ -}\} \\ & [x] \\ = \quad & \{\text{- definition of } gen_{ND} \text{ -}\} \\ & gen_{ND}\ x \end{aligned}$$

For the second equation, we proceed with a case analysis on the input.

case *Fail*

$$\begin{aligned} & extract_S\ (h'_{State}\ (alg\ Fail)) \\ = \quad & \{\text{- definition of } alg \text{ -}\} \\ & extract_S\ (h'_{State}\ pop_S) \\ = \quad & \{\text{- definition of } extract_S \text{ -}\} \\ & results \circ snd\ \$\ run_{State}\ (h'_{State}\ pop_S)\ (S\ [\,]\ [\,]) \\ = \quad & \{\text{- Lemma 7 -}\} \\ & results \circ snd\ \$\ ((), S\ [\,]\ [\,]) \\ = \quad & \{\text{- evaluation of } results, snd \text{ -}\} \\ & [\,] \\ = \quad & \{\text{- definition of } alg_{ND} \text{ -}\} \\ & alg_{ND}\ Fail \\ = \quad & \{\text{- definition of } fmap \text{ -}\} \\ & (alg_{ND} \circ fmap\ (extract_S \circ h'_{State}))\ Fail \end{aligned}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

case $Or\ p\ q$

$$
\begin{aligned}
& extract_S\ (h'_{State}\ (alg\ (Or\ p\ q))) \\
= \ & \{\text{- definition of } alg \text{ -}\} \\
& extract_S\ (h'_{State}\ (push_S\ q\ p)) \\
= \ & \{\text{- definition of } extract \text{ -}\} \\
& results \circ snd\ \$\ run_{State}\ (h'_{State}\ (push_S\ q\ p))\ (S\ [\,]\ [\,]) \\
= \ & \{\text{- Lemma 9 -}\} \\
& results \circ snd\ \$\ run_{State}\ (h'_{State}\ p)\ (S\ [\,]\ [q]) \\
= \ & \{\text{- Lemma 5 -}\} \\
& results \circ snd\ \$\ run_{State}\ (h'_{State}\ pop_S)\ (S\ ([\,] +\!\!+ extract_S\ (h'_{State}\ p))\ [q]) \\
= \ & \{\text{- definition of } (+\!\!+) \text{ -}\} \\
& results \circ snd\ \$\ run_{State}\ (h'_{State}\ pop_S)\ (S\ (extract_S\ (h'_{State}\ p))\ [q]) \\
= \ & \{\text{- Lemma 8 -}\} \\
& results \circ snd\ \$\ run_{State}\ (h'_{State}\ q)\ (S\ (extract_S\ (h'_{State}\ p))\ [\,]) \\
= \ & \{\text{- Lemma 5 -}\} \\
& results \circ snd\ \$\ run_{State}\ (h'_{State}\ pop_S)\ (S\ (extract_S\ (h'_{State}\ p) +\!\!+ extract_S\ (h'_{State}\ q))\ [\,]) \\
= \ & \{\text{- Lemma 7 -}\} \\
& results \circ snd\ \$\ ((), S\ (extract_S\ (h'_{State}\ p) +\!\!+ extract_S\ (h'_{State}\ q))\ [\,]) \\
= \ & \{\text{- evaluation of } results, snd \text{ -}\} \\
& extract_S\ (h'_{State}\ p) +\!\!+ extract_S\ (h'_{State}\ q) \\
= \ & \{\text{- definition of } alg_{ND} \text{ -}\} \\
& alg_{ND}\ (Or\ ((extract_S \circ h'_{State})\ p)\ ((extract_S \circ h'_{State})\ q)) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& (alg_{ND} \circ fmap\ (extract_S \circ h'_{State}))\ (Or\ p\ q)
\end{aligned}
$$

■

In the above proof we have used several lemmas. Now we prove them.

Lemma 5 (pop-extract)**.**

$$run_{State}\ (h'_{State}\ p)\ (S\ xs\ stack) = run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ p))\ stack)$$

holds for all p in the codomain of the function $nondet2state_S$.

Proof We prove this lemma by structural induction on $p :: Free\ (State_F\ (S\ a))\ ()$. For each inductive case of p, we not only assume this lemma holds for its sub-terms (this is the standard induction hypothesis) but also assume Theorem 2 holds for p and its sub-terms. This is sound because in the proof of Theorem 2, for $(extract_S \circ h'_{State} \circ nondet2state_S)\ p = h_{ND}\ p$, we only apply Lemma 5 to the sub-terms of p, which is already included in the induction hypothesis so there is no circular argument.

Since we assume Theorem 2 holds for p and its sub-terms, we can use several useful properties proved in the sub-cases of the proof of Theorem 2. We list them here for easy reference:

- extract-gen: $extract_S \circ h'_{State} \circ gen = \eta$
- extract-alg1: $extract_S\ (h'_{State}\ (alg\ Fail)) = [\,]$
- extract-alg2: $extract_S\ (h'_{State}\ (alg\ (Or\ p\ q))) = extract_S\ (h'_{State}\ p) +\!\!+ extract_S\ (h'_{State}\ q)$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

We proceed by structural induction on p. Note that for all p in the codomain of $nondet2state_S$, it is either generated by the gen or the alg of $nondet2state_S$. Thus, we only need to prove the following two equations where $p = gen\ x$ or $p = alg\ x$ and x is in the codomain of $fmap\ nondet2state_S$.

1. $run_{State}\ (h'_{State}\ (gen\ x))\ (S\ xs\ stack) = run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ (gen\ x)))\ stack)$
2. $run_{State}\ (h'_{State}\ (alg\ x))\ (S\ xs\ stack) = run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ (alg\ x)))\ stack)$

For the case $p = gen\ x$, we calculate as follows:

$$
\begin{aligned}
& run_{State}\ (h'_{State}\ (gen\ x))\ (S\ xs\ stack) \\
= \quad & \{\text{- definition of } gen \text{ -}\} \\
& run_{State}\ (h'_{State}\ (append_S\ x\ pop_S))\ (S\ xs\ stack) \\
= \quad & \{\text{- Lemma 6 -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ [x])\ stack) \\
= \quad & \{\text{- definition of } \eta \text{ -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ \eta\ x)\ stack) \\
= \quad & \{\text{- extract-gen -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ (gen\ x)))\ stack)
\end{aligned}
$$

For the case $p = alg\ x$, we proceed with a case analysis on x.

case *Fail*

$$
\begin{aligned}
& run_{State}\ (h'_{State}\ (alg\ Fail))\ (S\ xs\ stack) \\
= \quad & \{\text{- definition of } alg \text{ -}\} \\
& run_{State}\ (h'_{State}\ (pop_S))\ (S\ xs\ stack) \\
= \quad & \{\text{- definition of [ ] -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ [\,])\ stack) \\
= \quad & \{\text{- extract-alg1 -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ (alg\ Fail)))\ stack)
\end{aligned}
$$

case $Or\ p_1\ p_2$

$$
\begin{aligned}
& run_{State}\ (h'_{State}\ (alg\ (Or\ p_1\ p_2)))\ (S\ xs\ stack) \\
= \quad & \{\text{- definition of } alg \text{ -}\} \\
& run_{State}\ (h'_{State}\ (push_S\ p_2\ p_1))\ (S\ xs\ stack) \\
= \quad & \{\text{- Lemma 9 -}\} \\
& run_{State}\ (h'_{State}\ p_1)\ (S\ xs\ (p_2 : stack)) \\
= \quad & \{\text{- induction hypothesis -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ p_1))\ (p_2 : stack)) \\
= \quad & \{\text{- Lemma 8 -}\} \\
& run_{State}\ (h'_{State}\ p_2)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ p_1))\ stack) \\
= \quad & \{\text{- induction hypothesis -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ extract_S\ (h'_{State}\ p_1) +\!\!+ extract_S\ (h'_{State}\ p_2))\ stack) \\
= \quad & \{\text{- extract-alg2 -}\} \\
& run_{State}\ (h'_{State}\ pop_S)\ (S\ (xs +\!\!+ h'_{State}\ (alg\ (Or\ p_1\ p_2)))\ stack)
\end{aligned}
$$

■

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

The following four lemmas characterise the behaviours of stack operations.

Lemma 6 (evaluation-append)**.**

$$run_{State}\,(h'_{State}\,(append_S\,x\,p))\,(S\,xs\,stack) = run_{State}\,(h'_{State}\,p)\,(S\,(xs +\!\!+ [x])\,stack)$$

Proof

$$\begin{aligned}
&run_{State}\,(h'_{State}\,(append_S\,x\,p))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } append_S \text{ -}\}\\
&run_{State}\,(h'_{State}\,(get \gg\!= \lambda(S\,xs\,stack) \to put\,(S\,(xs +\!\!+ [x])\,stack) \gg p))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } get \text{ -}\}\\
&run_{State}\,(h'_{State}\,(Op\,(Get\,\eta) \gg\!= \lambda(S\,xs\,stack) \to put\,(S\,(xs +\!\!+ [x])\,stack) \gg p))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\}\\
&run_{State}\,(h'_{State}\,(Op\,(Get\,(\lambda(S\,xs\,stack) \to put\,(S\,(xs +\!\!+ [x])\,stack) \gg p))))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } h'_{State} \text{ -}\}\\
&run_{State}\,(State\,(\lambda s \to run_{State}\,(h'_{State}\,((\lambda(S\,xs\,stack) \to put\,(S\,(xs +\!\!+ [x])\,stack) \gg p)\,s))\,s))\\
&\quad(S\,xs\,stack)\\
=\;& \{\text{- definition of } run_{State} \text{ -}\}\\
&(\lambda s \to run_{State}\,(h'_{State}\,((\lambda(S\,xs\,stack) \to put\,(S\,(xs +\!\!+ [x])\,stack) \gg p)\,s))\,s)\,(S\,xs\,stack)\\
=\;& \{\text{- function application -}\}\\
&run_{State}\,(h'_{State}\,((\lambda(S\,xs\,stack) \to put\,(S\,(xs +\!\!+ [x])\,stack) \gg p)\,(S\,xs\,stack)))\,(S\,xs\,stack)\\
=\;& \{\text{- function application -}\}\\
&run_{State}\,(h'_{State}\,(put\,(S\,(xs +\!\!+ [x])\,stack) \gg p))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } put \text{ -}\}\\
&run_{State}\,(h'_{State}\,(Op\,(Put\,(S\,(xs +\!\!+ [x])\,stack)\,(\eta\,())) \gg p))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } (\gg) \text{ for free monad and Law 2.3: return-bind -}\}\\
&run_{State}\,(h'_{State}\,(Op\,(Put\,(S\,(xs +\!\!+ [x])\,stack)\,p)))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } h'_{State} \text{ -}\}\\
&run_{State}\,(State\,(\lambda s \to run_{State}\,(h'_{State}\,p)\,(S\,(xs +\!\!+ [x])\,stack)))\,(S\,xs\,stack)\\
=\;& \{\text{- definition of } run_{State} \text{ -}\}\\
&(\lambda s \to run_{State}\,(h'_{State}\,p)\,(S\,(xs +\!\!+ [x])\,stack))\,(S\,xs\,stack)\\
=\;& \{\text{- function application -}\}\\
&run_{State}\,(h'_{State}\,p)\,(S\,(xs +\!\!+ [x])\,stack)
\end{aligned}$$

■

Lemma 7 (evaluation-pop1)**.**

$$run_{State}\,(h'_{State}\,pop_S)\,(S\,xs\,[\,]) = ((), S\,xs\,[\,])$$

Proof

$$\begin{aligned}
&run_{State}\,(h'_{State}\,pop_S)\,(S\,xs\,[\,])\\
=\;& \{\text{- definition of } pop_S \text{ -}\}\\
&run_{State}\,(h'_{State}\,(get \gg\!= \lambda(S\,xs\,stack) \to
\end{aligned}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&\quad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op))\ (S\ xs\ [\,]) \\
&= \ \{\text{- definition of } get \text{ -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Get\ \eta) \gg\!= \lambda(S\ xs\ stack) \to \\
&\quad\ \ \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op))\ (S\ xs\ [\,]) \\
&= \ \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Get\ (\lambda(S\ xs\ stack) \to \\
&\quad\ \ \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op))))\ (S\ xs\ [\,]) \\
&= \ \{\text{- definition of } h'_{State} \text{ -}\} \\
&\quad run_{State}\ (State\ (\lambda s \to run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \to \\
&\quad\ \ \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op)\ s))\ s))\ (S\ xs\ [\,]) \\
&= \ \{\text{- definition of } run_{State} \text{ -}\} \\
&\quad (\lambda s \to run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \to \\
&\quad\ \ \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op)\ s))\ s)\ (S\ xs\ [\,]) \\
&= \ \{\text{- function application -}\} \\
&\quad run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \to \\
&\quad\ \ \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op)\ (S\ xs\ [\,])))\ (S\ xs\ [\,]) \\
&= \ \{\text{- function application, } \textbf{case}\text{-analysis -}\} \\
&\quad run_{State}\ (h'_{State}\ (\eta\ ()))\ (S\ xs\ [\,]) \\
&= \ \{\text{- definition of } h'_{State} \text{ -}\} \\
&\quad run_{State}\ (State\ (\lambda s \to ((), s)))\ (S\ xs\ [\,]) \\
&= \ \{\text{- definition of } run_{State}\text{, function application -}\} \\
&\quad ((), S\ xs\ [\,])
\end{aligned}
$$

■

Lemma 8 (evaluation-pop2)**.**

$$run_{State}\ (h'_{State}\ pop_S)\ (S\ xs\ (q : stack)) = run_{State}\ (h'_{State}\ q)\ (S\ xs\ stack)$$

Proof

$$
\begin{aligned}
&\quad run_{State}\ (h'_{State}\ pop_S)\ (S\ xs\ (q : stack)) \\
&= \ \{\text{- definition of } pop_S \text{ -}\} \\
&\quad run_{State}\ (h'_{State}\ (get \gg\!= \lambda(S\ xs\ stack) \to \\
&\quad\ \ \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op))\ (S\ xs\ (q : stack)) \\
&= \ \{\text{- definition of } get \text{ -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Get\ \eta) \gg\!= \lambda(S\ xs\ stack) \to \\
&\quad\ \ \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
&\qquad\qquad\quad\ op : ps \to \textbf{do}\ put\ (S\ xs\ ps); op))\ (S\ xs\ (q : stack))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Get\ (\lambda(S\ xs\ stack) \rightarrow \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad\ \rightarrow \eta\ () \\
&\qquad\qquad\quad op : ps \rightarrow \textbf{do}\ put\ (S\ xs\ ps);\ op))))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } h'_{State} \text{ -}\} \\
&\quad run_{State}\ (State\ (\lambda s \rightarrow run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \rightarrow \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad\ \rightarrow \eta\ () \\
&\qquad\qquad\quad op : ps \rightarrow \textbf{do}\ put\ (S\ xs\ ps);\ op)\ s))\ s))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } run_{State} \text{ -}\} \\
&\quad (\lambda s \rightarrow run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \rightarrow \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad\ \rightarrow \eta\ () \\
&\qquad\qquad\quad op : ps \rightarrow \textbf{do}\ put\ (S\ xs\ ps);\ op)\ s))\ s)\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \rightarrow \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad\ \rightarrow \eta\ () \\
&\qquad\qquad\quad op : ps \rightarrow \textbf{do}\ put\ (S\ xs\ ps);\ op)\ (S\ xs\ (q : stack))))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- function application, } \textbf{case}\text{-analysis -}\}
\end{aligned}
$$

$$
\begin{aligned}
&\quad run_{State}\ (h'_{State}\ (put\ (S\ xs\ stack) \gg q))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } put \text{ -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Put\ (S\ xs\ stack)\ (\eta\ ())) \gg q))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } (\gg) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Put\ (S\ xs\ stack)\ q)))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } h'_{State} \text{ -}\} \\
&\quad run_{State}\ (State\ (\lambda s \rightarrow run_{State}\ (h'_{State}\ q)\ (S\ xs\ stack)))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } run_{State} \text{ -}\} \\
&\quad (\lambda s \rightarrow run_{State}\ (h'_{State}\ q)\ (S\ xs\ stack))\ (S\ xs\ (q : stack)) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{State}\ (h'_{State}\ q)\ (S\ xs\ stack)
\end{aligned}
$$

■

Lemma 9 (evaluation-push)**.**

$$run_{State}\ (h'_{State}\ (push_S\ q\ p))\ (S\ xs\ stack) = run_{State}\ (h'_{State}\ p)\ (S\ xs\ (q : stack))$$

Proof

$$
\begin{aligned}
&\quad run_{State}\ (h'_{State}\ (push_S\ q\ p))\ (S\ xs\ stack) \\
&= \quad \{\text{- definition of } push_S \text{ -}\}
\end{aligned}
$$

$$
\begin{aligned}
&\quad run_{State}\ (h'_{State}\ (get \gg\!= \lambda(S\ xs\ stack) \rightarrow put\ (S\ xs\ (q : stack)) \gg p))\ (S\ xs\ stack) \\
&= \quad \{\text{- definition of } get \text{ -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Get\ \eta) \gg\!= \lambda(S\ xs\ stack) \rightarrow put\ (S\ xs\ (q : stack)) \gg p))\ (S\ xs\ stack) \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Get\ (\lambda(S\ xs\ stack) \rightarrow put\ (S\ xs\ (q : stack)) \gg p))))\ (S\ xs\ stack)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } h'_{State} \text{ -}\} \\
&\quad run_{State}\ (State\ (\lambda s \to run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \to put\ (S\ xs\ (q : stack)) \gg p)\ s))\ s))\\
&\qquad (S\ xs\ stack) \\
&= \quad \{\text{- definition of } run_{State} \text{ -}\} \\
&\quad (\lambda s \to run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \to put\ (S\ xs\ (q : stack)) \gg p)\ s))\ s)\ (S\ xs\ stack) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{State}\ (h'_{State}\ ((\lambda(S\ xs\ stack) \to put\ (S\ xs\ (q : stack)) \gg p)\ (S\ xs\ stack)))\ (S\ xs\ stack) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{State}\ (h'_{State}\ (put\ (S\ xs\ (q : stack)) \gg p))\ (S\ xs\ stack) \\
&= \quad \{\text{- definition of } put \text{ -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Put\ (S\ xs\ (q : stack))\ (\eta\ ())) \gg p))\ (S\ xs\ stack) \\
&= \quad \{\text{- definition of } (\gg) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{State}\ (h'_{State}\ (Op\ (Put\ (S\ xs\ (q : stack))\ p)))\ (S\ xs\ stack) \\
&= \quad \{\text{- definition of } h'_{State} \text{ -}\} \\
&\quad run_{State}\ (State\ (\lambda s \to run_{State}\ (h'_{State}\ p)\ (S\ xs\ (q : stack))))\ (S\ xs\ stack) \\
&= \quad \{\text{- definition of } run_{State} \text{ -}\} \\
&\quad (\lambda s \to run_{State}\ (h'_{State}\ p)\ (S\ xs\ (q : stack)))\ (S\ xs\ stack) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{State}\ (h'_{State}\ p)\ (S\ xs\ (q : stack))
\end{aligned}
$$

■

3.2 Combining with other effects

This section proves the following theorem in Section 5.2.

Theorem 3. $run_{ND+f} = h_{ND+f}$

Proof The proof is very similar to that of Theorem 2 in Appendix 3.1.

We start with expanding the definition of run_{ND+f}:

$$extract_{SS} \circ h_{State} \circ nondet2state = h_{ND}$$

We use the fold fusion law **fusion-post'** (3.3) to fuse the left-hand side. Since the right-hand side is already a fold, to prove the equation we just need to check the components of the fold h_{ND} satisfy the conditions of the fold fusion. The conditions can be splitted into the following three equations:

$$
\begin{aligned}
& (extract_{SS} \circ h_{State}) \circ gen = gen_{ND+f} \\
& (extract_{SS} \circ h_{State}) \circ alg \circ fmap\ nondet2state_S \\
= \quad & alg_{ND+f} \circ fmap\ (extract_{SS} \circ h_{State}) \circ fmap\ nondet2state_S \\
& (extract_{SS} \circ h_{State}) \circ fwd \circ fmap\ nondet2state_S \\
= \quad & fwd_{ND+f} \circ fmap\ (extract_{SS} \circ h_{State}) \circ fmap\ nondet2state_S
\end{aligned}
$$

For brevity, we omit the last common part $fmap\ nondet2state_S$ of the second equation in the following proof. Instead, we assume that the input is in the codomain of $fmap\ nondet2state_S$.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

For the first equation, we calculate as follows:

$$
\begin{aligned}
& \mathit{extract}_{SS}\ (h_{State}\ (\mathit{gen}\ x)) \\
= \quad & \{\text{- definition of } \mathit{gen} \text{ -}\} \\
& \mathit{extract}_{SS}\ (h_{State}\ (\mathit{append}_{SS}\ x\ \mathit{pop}_{SS})) \\
= \quad & \{\text{- definition of } \mathit{extract}_{SS} \text{ -}\} \\
& \mathit{results}_{SS} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{StateT}\ (h_{State}\ (\mathit{append}_{SS}\ x\ \mathit{pop}_{SS}))\ (SS\ [\,]\ [\,]) \\
= \quad & \{\text{- Lemma 11 -}\} \\
& \mathit{results}_{SS} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{StateT}\ (h_{State}\ \mathit{pop}_{SS})\ (SS\ ([\,] +\!\!+ [x])\ [\,]) \\
= \quad & \{\text{- definition of } (+\!\!+) \text{ -}\} \\
& \mathit{results}_{SS} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{StateT}\ (h_{State}\ \mathit{pop}_{SS})\ (SS\ [x]\ [\,]) \\
= \quad & \{\text{- Lemma 12 -}\} \\
& \mathit{results}_{SS} \circ \mathit{snd}\ \langle\$\rangle\ \eta\ ((), SS\ [x]\ [\,]) \\
= \quad & \{\text{- evaluation of } \mathit{snd}, \mathit{results}_{SS} \text{ -}\} \\
& \eta\ [x] \\
= \quad & \{\text{- definition of } \eta \text{ for free monad -}\} \\
& \mathit{Var}\ [x] \\
= \quad & \{\text{- definition of } \eta_{[]} \text{ -}\} \\
& (\mathit{Var} \circ \eta)\ x \\
= \quad & \{\text{- definition of } \mathit{gen}_{ND+f} \text{ -}\} \\
& \mathit{gen}_{ND+f}\ x
\end{aligned}
$$

For the second equation, we proceed with a case analysis on the input.

case *Fail*

$$
\begin{aligned}
& \mathit{extract}_{SS}\ (h_{State}\ (\mathit{alg}\ \mathit{Fail})) \\
= \quad & \{\text{- definition of } \mathit{alg} \text{ -}\} \\
& \mathit{extract}_{SS}\ (h_{State}\ \mathit{pop}_{SS}) \\
= \quad & \{\text{- definition of } \mathit{extract}_{SS} \text{ -}\} \\
& \mathit{results}_{SS} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{StateT}\ (h_{State}\ \mathit{pop}_{SS})\ (SS\ [\,]\ [\,]) \\
= \quad & \{\text{- Lemma 12 -}\} \\
& \mathit{results}_{SS} \circ \mathit{snd}\ \langle\$\rangle\ \eta\ ((), SS\ [\,]\ [\,]) \\
= \quad & \{\text{- evaluation of } \mathit{snd}, \mathit{results}_{SS} \text{ -}\} \\
& \eta\ [\,] \\
= \quad & \{\text{- definition of } \eta \text{ for free monad -}\} \\
& \mathit{Var}\ [\,] \\
= \quad & \{\text{- definition of } \mathit{alg}_{ND+f} \text{ -}\} \\
& \mathit{alg}_{ND+f}\ \mathit{Fail} \\
= \quad & \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
& (\mathit{alg}_{ND+f} \circ \mathit{fmap}\ (\mathit{extract}_{SS} \circ h_{State}))\ \mathit{Fail}
\end{aligned}
$$

case *Inl* (*Or p q*)

$$
\begin{aligned}
& \mathit{extract}_{SS}\ (h_{State}\ (\mathit{alg}\ (\mathit{Or}\ p\ q))) \\
= \quad & \{\text{- definition of } \mathit{alg} \text{ -}\} \\
& \mathit{extract}_{SS}\ (h_{State}\ (\mathit{push}_{SS}\ q\ p)) \\
= \quad & \{\text{- definition of } \mathit{extract}_{SS} \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& results_{SS} \circ snd\ \langle\$\rangle\ run_{StateT}\ (h_{State}\ (push_{SS}\ q\ p))\ (SS\ [\,]\ [\,]) \\
= \quad & \{\text{- Lemma 14 -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle\ run_{StateT}\ (h_{State}\ p)\ (SS\ [\,]\ [q]) \\
= \quad & \{\text{- Lemma 10 -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle \\
& \quad \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p);\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ ([\,] +\!\!+ p')\ [q])\} \\
= \quad & \{\text{- definition of } (+\!\!+) \text{ -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle\ \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p);\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ p'\ [q])\} \\
= \quad & \{\text{- Lemma 13 -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle\ \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p);\ run_{StateT}\ (h_{State}\ q)\ (SS\ p'\ [\,])\} \\
= \quad & \{\text{- Lemma 10 -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle\ \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p); \\
& \quad \mathbf{do}\ \{q' \leftarrow extract_{SS}\ (h_{State}\ q);\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (p' +\!\!+ q')\ [\,])\}\} \\
= \quad & \{\text{- Law (2.5) for } \mathbf{do}\text{-notation -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle\ \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p);\ q' \leftarrow extract_{SS}\ (h_{State}\ q); \\
& \quad run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (p' +\!\!+ q')\ [\,])\} \\
= \quad & \{\text{- Lemma 12 -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle \\
& \quad \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p);\ q' \leftarrow extract_{SS}\ (h_{State}\ q);\ \eta\ ((), SS\ (p' +\!\!+ q')\ [\,])\} \\
= \quad & \{\text{- evaluation of } snd, results_{SS} \text{ -}\} \\
& \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p);\ q' \leftarrow extract_{SS}\ (h_{State}\ q);\ \eta\ (p' +\!\!+ q')\} \\
= \quad & \{\text{- definition of } liftM2 \text{ -}\} \\
& liftM2\ (+\!\!+)\ ((extract_{SS} \circ h_{State})\ p)\ ((extract_{SS} \circ h_{State})\ q) \\
= \quad & \{\text{- definition of } alg_{ND+f} \text{ -}\} \\
& alg_{ND+f}\ (Or\ ((extract_{SS} \circ h_{State})\ p)\ ((extract_{SS} \circ h_{State})\ q)) \\
= \quad & \{\text{- definition of } fmap \text{ -}\} \\
& (alg_{ND+f} \circ fmap\ (extract_{SS} \circ h_{State}))\ (Or\ p\ q)
\end{aligned}
$$

For the last equation, we calculate as follows:

$$
\begin{aligned}
& extract_{SS}\ (h_{State}\ (fwd\ y)) \\
= \quad & \{\text{- definition of } fwd \text{ -}\} \\
& extract_{SS}\ (h_{State}\ (Op\ (Inr\ y))) \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& extract_{SS}\ (StateT\ \$\ \lambda s \rightarrow Op\ \$\ fmap\ (\lambda k \rightarrow run_{StateT}\ k\ s)\ (fmap\ h_{State}\ y)) \\
= \quad & \{\text{- definition of } extract_{SS} \text{ -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle \\
& \quad run_{StateT}\ (StateT\ \$\ \lambda s \rightarrow Op\ \$\ fmap\ (\lambda k \rightarrow run_{StateT}\ k\ s)\ (fmap\ h_{State}\ y))\ (SS\ [\,]\ [\,]) \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle\ (\lambda s \rightarrow Op\ \$\ fmap\ (\lambda k \rightarrow run_{StateT}\ k\ s)\ (fmap\ h_{State}\ y))\ (SS\ [\,]\ [\,]) \\
= \quad & \{\text{- function application -}\} \\
& results_{SS} \circ snd\ \langle\$\rangle\ Op\ \$\ fmap\ (\lambda k \rightarrow run_{StateT}\ k\ (SS\ [\,]\ [\,]))\ (fmap\ h_{State}\ y)) \\
= \quad & \{\text{- definition of } \langle\$\rangle \text{ -}\} \\
& Op\ (fmap\ (\lambda k \rightarrow results_{SS} \circ snd\ \langle\$\rangle\ run_{StateT}\ k\ (SS\ [\,]\ [\,]))\ (fmap\ h_{State}\ y)) \\
= \quad & \{\text{- definition of } fwd_{ND+f} \text{ -}\} \\
& fwd_{ND+f}\ (fmap\ (\lambda k \rightarrow results_{SS} \circ snd\ \langle\$\rangle\ run_{StateT}\ k\ (SS\ [\,]\ [\,]))\ (fmap\ h_{State}\ y))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } extract_{SS} \text{ -}\} \\
&\quad fwd_{ND+f}\ (fmap\ extract_{SS}\ (fmap\ h_{State}\ y)) \\
&= \quad \{\text{- Law (2.2) -}\} \\
&\quad fwd_{ND+f}\ (fmap\ (extract_{SS} \circ h_{State})\ y)
\end{aligned}
$$

■

In the above proof we have used several lemmas. Now we prove them.

Lemma 10 (pop-extract of *SS*)**.**

$$
\begin{aligned}
&run_{StateT}\ (h_{State}\ p)\ (SS\ xs\ stack) \\
&= \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ p);\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

holds for all p in the codomain of the function nondet2state.

Proof The proof structure is similar to that of Lemma 5. We prove this lemma by structural induction on $p :: Free\ (State_F\ (SS\ f\ a) :+: f)\ ()$. For each inductive case of p, we not only assume this lemma holds for its sub-terms (this is the standard induction hypothesis) but also assume Theorem 3 holds for p and its sub-terms. This is sound because in the proof of Theorem 3, for $(extract_{SS} \circ h_{State} \circ nondet2state)\ p = h_{ND+f}\ p$, we only apply Lemma 10 to the sub-terms of p, which is already included in the induction hypothesis so there is no circular argument.

Since we assume Theorem 3 holds for p and its sub-terms, we can use several useful properties proved in the sub-cases of the proof of Theorem 3. We list them here for easy reference:

- extract-gen-ext: $extract_{SS} \circ h_{State} \circ gen = Var \circ \eta$
- extract-alg1-ext: $extract_{SS}\ (h_{State}\ (alg\ Fail)) = Var\ [\,]$
- extract-alg2-ext: $extract_{SS}\ (h_{State}\ (alg\ (Or\ p\ q))) = liftM2\ (+\!\!+)\ (extract_{SS}\ (h_{State}\ p))\ (extract_{SS}\ (h_{State}\ q))$
- extract-fwd $extract_{SS}\ (h_{State}\ (fwd\ y)) = fwd_{ND+f}\ (fmap\ (extract_{SS} \circ h_{State})\ y)$

We proceed by structural induction on p. Note that for all p in the codomain of *nondet2state*, it is either generated by the *gen*, *alg*, or *fwd* of *nondet2state*. Thus, we only need to prove the following three equations, where x is in the codomain of $fmap\ nondet2state_S$ and $p = gen\ x$, $p = alg\ x$, and $p = fwd\ x$, respectively.

1. $$
\begin{aligned}
&run_{StateT}\ (h_{State}\ (gen\ x))\ (SS\ xs\ stack) \\
&= \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ (gen\ x));\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

2. $$
\begin{aligned}
&run_{StateT}\ (h_{State}\ (alg\ x))\ (SS\ xs\ stack) \\
&= \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ (alg\ x));\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

3. $$
\begin{aligned}
&run_{StateT}\ (h_{State}\ (fwd\ x))\ (SS\ xs\ stack) \\
&= \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ (fwd\ x));\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

For the case $p = gen\ x$, we calculate as follows:

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ (gen\ x))\ (SS\ xs\ stack) \\
= \quad & \{\text{- definition of } gen \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (append_{SS}\ x\ pop_{SS}))\ (SS\ xs\ stack) \\
= \quad & \{\text{- Lemma 11 -}\} \\
& run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ [x])\ stack) \\
= \quad & \{\text{- definition of } \eta_{[]} \text{ -}\} \\
& run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ \eta\ x)\ stack) \\
= \quad & \{\text{- definition of } Var \text{ and reformulation -}\} \\
& \mathbf{do}\ \{p' \leftarrow Var\ (\eta\ x);\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\} \\
= \quad & \{\text{- extract-gen-ext -}\} \\
& \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ (gen\ x));\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

For the case $p = alg\ x$, we proceed with a case analysis on x.

case *Fail*

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ (alg\ Fail))\ (SS\ xs\ stack) \\
= \quad & \{\text{- definition of } alg \text{ -}\} \\
& run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ xs\ stack) \\
= \quad & \{\text{- definition of } [\,] \text{ -}\} \\
& run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ [\,])\ stack) \\
= \quad & \{\text{- definition of } Var \text{ -}\} \\
& \mathbf{do}\ \{p' \leftarrow Var\ [\,];\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\} \\
= \quad & \{\text{- extract-alg1-ext -}\} \\
& \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ (alg\ Fail));\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

case $Or\ p_1\ p_2$

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ (alg\ (Or\ p_1\ p_2)))\ (SS\ xs\ stack) \\
= \quad & \{\text{- definition of } alg \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (push_{SS}\ p_2\ p_1))\ (SS\ xs\ stack) \\
= \quad & \{\text{- Lemma 14 -}\} \\
& run_{StateT}\ (h_{State}\ p_1)\ (SS\ xs\ (p_2 : stack)) \\
= \quad & \{\text{- induction hypothesis: pop-extract of } p_1 \text{ -}\} \\
& \mathbf{do}\ \{p_1' \leftarrow extract_{SS}\ (h_{State}\ p_1);\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p_1')\ (p_2 : stack))\} \\
= \quad & \{\text{- Lemma 13 -}\} \\
& \mathbf{do}\ \{p_1' \leftarrow extract_{SS}\ (h_{State}\ p_1);\ run_{StateT}\ (h_{State}\ p_2)\ (SS\ (xs +\!\!+ p_1')\ stack)\} \\
= \quad & \{\text{- induction hypothesis: pop-extract of } p_2 \text{ -}\} \\
& \mathbf{do}\ \{p_2' \leftarrow extract_{SS}\ (h_{State}\ p_2);\ \mathbf{do}\ \{p_1' \leftarrow extract_{SS}\ (h_{State}\ p_1); \\
& \quad run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p_1' +\!\!+ p_2')\ stack)\}\} \\
= \quad & \{\text{- Law (2.5) with } \mathbf{do}\text{-notation -}\} \\
& \mathbf{do}\ \{p_2' \leftarrow extract_{SS}\ (h_{State}\ p_2);\ p_1' \leftarrow extract_{SS}\ (h_{State}\ p_1); \\
& \quad run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p_1' +\!\!+ p_2')\ stack)\} \\
= \quad & \{\text{- definition of } liftM2 \text{ -}\} \\
& \mathbf{do}\ \{p' \leftarrow liftM2\ (+\!\!+)\ (extract_{SS}\ (h_{State}\ p_2))\ (extract_{SS}\ (h_{State}\ p_1)); \\
& \quad run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\} \\
= \quad & \{\text{- extract-alg2-ext -}\} \\
& \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ (alg\ (Or\ p_1\ p_2)));\ run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

For the case $p = fwd\ x$, we proceed with a case analysis on x.

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ (fwd\ x))\ (SS\ xs\ stack) \\
= \ & \{\text{- definition of } fwd \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (Op\ (Inr\ x)))\ (SS\ xs\ stack) \\
= \ & \{\text{- definition of } h_{State} \text{ -}\} \\
& run_{StateT}\ (StateT\ (\lambda s \to Op\ (fmap\ (\lambda y \to run_{StateT}\ (h_{State}\ y\ s))\ x)))\ (SS\ xs\ stack) \\
= \ & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& Op\ (fmap\ (\lambda y \to run_{StateT}\ (h_{State}\ y\ (SS\ xs\ stack)))\ x) \\
= \ & \{\text{- induction hypothesis -}\} \\
& Op\ (fmap\ (\lambda y \to \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ y); run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\})\ x) \\
= \ & \{\text{- definition of } \gg\!= \text{ -}\} \\
& \mathbf{do}\ \{p' \leftarrow Op\ (fmap\ (\lambda y \to extract_{SS}\ (h_{State}\ y))\ x); run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\} \\
= \ & \{\text{- definition of } fwd_{ND+f} \text{ -}\} \\
& \mathbf{do}\ \{p' \leftarrow fwd_{ND+f}\ (fmap\ (\lambda y \to extract_{SS}\ (h_{State}\ y))\ x); run_{StateT}\ (h_{State}\ pop_{SS}) \\
& (SS\ (xs +\!\!+ p')\ stack)\} \\
= \ & \{\text{- extract-fwd -}\} \\
= \ & \mathbf{do}\ \{p' \leftarrow extract_{SS}\ (h_{State}\ (fwd\ x)); run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ (xs +\!\!+ p')\ stack)\}
\end{aligned}
$$

■

The following four lemmas characterize the behaviours of stack operations.

Lemma 11 (evaluation-append-ext)**.**

$$run_{StateT}\ (h_{State}\ (append_{SS}\ x\ p))\ (SS\ xs\ stack) = run_{StateT}\ (h_{State}\ p)\ (SS\ (xs +\!\!+ [x])\ stack)$$

Proof

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ (append_{SS}\ x\ p))\ (SS\ xs\ stack) \\
= \ & \{\text{- definition of } append_{SS} \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (get \gg\!= \lambda(SS\ xs\ stack) \to put\ (SS\ (xs +\!\!+ [x])\ stack) \gg p))\ (SS\ xs\ stack) \\
= \ & \{\text{- definition of } get \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ \eta)) \gg\!= \lambda(SS\ xs\ stack) \to put\ (SS\ (xs +\!\!+ [x])\ stack) \gg p)) \\
& (SS\ xs\ stack) \\
= \ & \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\} \\
& run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ (\lambda(SS\ xs\ stack) \to put\ (SS\ (xs +\!\!+ [x])\ stack) \gg p))))) \\
& (SS\ xs\ stack) \\
= \ & \{\text{- definition of } h_{State} \text{ -}\} \\
& run_{StateT}\ (StateT\ (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to put\ (SS\ (xs +\!\!+ [x])\ stack) \\
& \gg p)\ s))\ s))\ (SS\ xs\ stack) \\
= \ & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to put\ (SS\ (xs +\!\!+ [x])\ stack) \gg p)\ s))\ s) \\
& (SS\ xs\ stack) \\
= \ & \{\text{- function application -}\} \\
& run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to put\ (SS\ (xs +\!\!+ [x])\ stack) \gg p)\ (SS\ xs\ stack))) \\
& (SS\ xs\ stack)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- function application -}\} \\
&\quad run_{StateT}\ (h_{State}\ (put\ (SS\ (xs +\!\!+ [x])\ stack) \gg p))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } put \text{ -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ (SS\ (xs +\!\!+ [x])\ stack)\ (\eta\ ()))) \gg p))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } (\gg) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ (SS\ (xs +\!\!+ [x])\ stack)\ p))))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } h_{State} \text{ -}\} \\
&\quad run_{StateT}\ (StateT\ (\lambda s \to run_{StateT}\ (h_{State}\ p)\ (SS\ (xs +\!\!+ [x])\ stack)))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } run_{StateT} \text{ -}\} \\
&\quad (\lambda s \to run_{StateT}\ (h_{State}\ p)\ (SS\ (xs +\!\!+ [x])\ stack))\ (SS\ xs\ stack) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{StateT}\ (h_{State}\ p)\ (SS\ (xs +\!\!+ [x])\ stack)
\end{aligned}
$$

■

Lemma 12 (evaluation-pop1-ext)**.**

$$run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ xs\ [\,]) = \eta\ ((), SS\ xs\ [\,])$$

Proof

$$
\begin{aligned}
&\quad run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ xs\ [\,]) \\
&= \quad \{\text{- definition of } pop_{SS} \text{ -}\} \\
&\quad run_{StateT}\ (h_{State}\ (get \gg\!= \lambda(SS\ xs\ stack) \to \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \qquad \to \eta\ () \\
&\qquad\qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps); op))\ (SS\ xs\ [\,]) \\
&= \quad \{\text{- definition of } get \text{ -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ \eta)) \gg\!= \lambda(SS\ xs\ stack) \to \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \qquad \to \eta\ () \\
&\qquad\qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps); op))\ (SS\ xs\ [\,]) \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ (\lambda(SS\ xs\ stack) \to \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \qquad \to \eta\ () \\
&\qquad\qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps); op)))))\ (SS\ xs\ [\,]) \\
&= \quad \{\text{- definition of } h_{State} \text{ -}\} \\
&\quad run_{StateT}\ (StateT\ (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \qquad \to \eta\ () \\
&\qquad\qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps); op)\ s))\ s))\ (SS\ xs\ [\,]) \\
&= \quad \{\text{- definition of } run_{StateT} \text{ -}\} \\
&\quad (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \qquad \to \eta\ () \\
&\qquad\qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps); op)\ s))\ s)\ (SS\ xs\ [\,]) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to \\
&\qquad \textbf{case}\ stack\ \textbf{of}\ [\,] \qquad \to \eta\ () \\
&\qquad\qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps); op)\ (SS\ xs\ [\,])))\ (SS\ xs\ [\,]) \\
&= \quad \{\text{- function application, } \textbf{case}\text{-analysis -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ (\eta\ ()))\ (SS\ xs\ [\,]) \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& run_{StateT}\ (StateT\ (\lambda s \to \eta\ ((), s)))\ (SS\ xs\ [\,]) \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& (\lambda s \to \eta\ ((), s))\ (SS\ xs\ [\,]) \\
= \quad & \{\text{- function application -}\} \\
& ((), SS\ xs\ [\,])
\end{aligned}
$$

■

Lemma 13 (evaluation-pop2-ext)**.**

$$run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ xs\ (q : stack)) = run_{StateT}\ (h_{State}\ q)\ (SS\ xs\ stack)$$

Proof

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ pop_{SS})\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- definition of } pop_{SS} \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (get \gg\!= \lambda(SS\ xs\ stack) \to \\
& \quad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
& \qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps);\ op))\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- definition of } get \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ \eta)) \gg\!= \lambda(SS\ xs\ stack) \to \\
& \quad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
& \qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps);\ op))\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\} \\
& run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ (\lambda(SS\ xs\ stack) \to \\
& \quad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
& \qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps);\ op)))))\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& run_{StateT}\ (StateT\ (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to \\
& \quad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
& \qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps);\ op)\ s))\ s))\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to \\
& \quad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
& \qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps);\ op)\ s))\ s)\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- function application -}\} \\
& run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to \\
& \quad \textbf{case}\ stack\ \textbf{of}\ [\,] \quad \to \eta\ () \\
& \qquad\qquad op : ps \to \textbf{do}\ put\ (SS\ xs\ ps);\ op)\ (SS\ xs\ (q : stack))))\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- function application, } \textbf{case}\text{-analysis -}\}
\end{aligned}
$$

$$
\begin{aligned}
& run_{StateT}\ (h_{State}\ (put\ (SS\ xs\ stack) \gg q))\ (SS\ xs\ (q : stack)) \\
= \quad & \{\text{- definition of } put \text{ -}\} \\
& run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ (SS\ xs\ stack)\ (\eta\ ()))) \gg q))\ (SS\ xs\ (q : stack))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- definition of } (\gg) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ (SS\ xs\ stack)\ q))))\ (SS\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } h_{State} \text{ -}\} \\
&\quad run_{StateT}\ (StateT\ (\lambda s \to run_{StateT}\ (h_{State}\ q)\ (SS\ xs\ stack)))\ (SS\ xs\ (q : stack)) \\
&= \quad \{\text{- definition of } run_{StateT} \text{ -}\} \\
&\quad (\lambda s \to run_{StateT}\ (h_{State}\ q)\ (SS\ xs\ stack))\ (SS\ xs\ (q : stack)) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{StateT}\ (h_{State}\ q)\ (SS\ xs\ stack)
\end{aligned}
$$

■

Lemma 14 (evaluation-push-ext)**.**

$$run_{StateT}\ (h_{State}\ (push_{SS}\ q\ p))\ (SS\ xs\ stack) = run_{StateT}\ (h_{State}\ p)\ (SS\ xs\ (q : stack))$$

Proof

$$
\begin{aligned}
&run_{StateT}\ (h_{State}\ (push_{SS}\ q\ p))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } push_{SS} \text{ -}\} \\
&\quad run_{StateT}\ (h_{State}\ (get \gg\!= \lambda(SS\ xs\ stack) \to put\ (SS\ xs\ (q : stack)) \gg p))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } get \text{ -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ \eta)) \gg\!= \lambda(SS\ xs\ stack) \to put\ (SS\ xs\ (q : stack)) \gg p))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ (\lambda(SS\ xs\ stack) \to put\ (SS\ xs\ (q : stack)) \gg p)))))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } h_{State} \text{ -}\} \\
&\quad run_{StateT}\ (StateT\ (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to put\ (SS\ xs\ (q : stack)) \gg p)\ s))\ s))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } run_{StateT} \text{ -}\} \\
&\quad (\lambda s \to run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to put\ (SS\ xs\ (q : stack)) \gg p)\ s))\ s)\ (SS\ xs\ stack) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{StateT}\ (h_{State}\ ((\lambda(SS\ xs\ stack) \to put\ (SS\ xs\ (q : stack)) \gg p)\ (SS\ xs\ stack)))\ (SS\ xs\ stack) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{StateT}\ (h_{State}\ (put\ (SS\ xs\ (q : stack)) \gg p))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } put \text{ -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ (SS\ xs\ (q : stack))\ (\eta\ ()))) \gg p))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } (\gg) \text{ for free monad and Law 2.3: return-bind -}\} \\
&\quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ (SS\ xs\ (q : stack))\ p))))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } h_{State} \text{ -}\} \\
&\quad run_{StateT}\ (StateT\ (\lambda s \to run_{StateT}\ (h_{State}\ p)\ (SS\ xs\ (q : stack))))\ (SS\ xs\ stack) \\
&= \quad \{\text{- definition of } run_{StateT} \text{ -}\} \\
&\quad (\lambda s \to run_{StateT}\ (h_{State}\ p)\ (SS\ xs\ (q : stack)))\ (SS\ xs\ stack) \\
&= \quad \{\text{- function application -}\} \\
&\quad run_{StateT}\ (h_{State}\ p)\ (SS\ xs\ (q : stack))
\end{aligned}
$$

■

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

4 Proofs for modelling two states with one state

In this section, we prove the following theorem in Section 6.1.

Theorem 4. $h_{States} = nest \circ h_{State} \circ states2state$

Proof Instead of proving it directly, we show the correctness of the isomorphism of *nest* and *flatten* and prove the following equation:

$$flatten \circ h_{States} = h_{State} \circ states2state$$

It is obvious that α and α^{-1} form an isomorphism, i.e., $\alpha \circ \alpha^{-1} = id$ and $\alpha^{-1} \circ \alpha = id$. We show that *nest* and *flatten* form an isomorphism by calculation. For all $t :: StateT\ s_1\ (StateT\ s_2\ (Free\ f))\ a$, we show that $(nest \circ flatten)\ t = t$.

$$\begin{aligned}
&(nest \circ flatten)\ t \\
=\ &\{\text{- definition of } flatten \text{ -}\} \\
&nest\ \$\ StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ run_{StateT}\ (run_{StateT}\ t\ s_1)\ s_2 \\
=\ &\{\text{- definition of } nest \text{ -}\} \\
&StateT\ \$\ \lambda s_1 \to StateT\ \$\ \lambda s_2 \to \alpha^{-1}\ \langle\$\rangle \\
&\quad run_{StateT}\ (StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ run_{StateT}\ (run_{StateT}\ t\ s_1)\ s_2)\ (s_1, s_2) \\
=\ &\{\text{- definition of } run_{StateT} \text{ -}\} \\
&StateT\ \$\ \lambda s_1 \to StateT\ \$\ \lambda s_2 \to \alpha^{-1}\ \langle\$\rangle \\
&\quad (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ run_{StateT}\ (run_{StateT}\ t\ s_1)\ s_2)\ (s_1, s_2) \\
=\ &\{\text{- function application -}\} \\
&StateT\ \$\ \lambda s_1 \to StateT\ \$\ \lambda s_2 \to \alpha^{-1}\ \langle\$\rangle\ (\alpha\ \langle\$\rangle\ run_{StateT}\ (run_{StateT}\ t\ s_1)\ s_2) \\
=\ &\{\text{- Equation (2.2) -}\} \\
&StateT\ \$\ \lambda s_1 \to StateT\ \$\ \lambda s_2 \to (fmap\ (\alpha^{-1} \circ \alpha)\ (run_{StateT}\ (run_{StateT}\ t\ s_1)\ s_2)) \\
=\ &\{\text{- } \alpha^{-1} \circ \alpha = id \text{ -}\} \\
&StateT\ \$\ \lambda s_1 \to StateT\ \$\ \lambda s_2 \to (fmap\ id\ (run_{StateT}\ (run_{StateT}\ t\ s_1)\ s_2)) \\
=\ &\{\text{- Equation (2.1) -}\} \\
&StateT\ \$\ \lambda s_1 \to StateT\ \$\ \lambda s_2 \to (run_{StateT}\ (run_{StateT}\ t\ s_1)\ s_2) \\
=\ &\{\text{- } \eta\text{-reduction and reformulation -}\} \\
&StateT\ \$\ \lambda s_1 \to (StateT \circ run_{StateT})\ (run_{StateT}\ t\ s_1) \\
=\ &\{\text{- } StateT \circ run_{StateT} = id \text{ -}\} \\
&StateT\ \$\ \lambda s_1 \to run_{StateT}\ t\ s_1 \\
=\ &\{\text{- } \eta\text{-reduction -}\} \\
&StateT\ \$\ run_{StateT}\ t \\
=\ &\{\text{- } StateT \circ run_{StateT} = id \text{ -}\} \\
&t
\end{aligned}$$

For all $t :: StateT\ (s_1, s_2)\ (Free\ f)\ a$, we show that $(flatten \circ nest)\ t = t$.

$$\begin{aligned}
&(flatten \circ nest)\ t = t \\
=\ &\{\text{- definition of } nest \text{ -}\} \\
&flatten\ \$\ StateT\ \$\ \lambda s_1 \to StateT\ \$\ \lambda s_2 \to \alpha^{-1}\ \langle\$\rangle\ run_{StateT}\ t\ (s_1, s_2) \\
=\ &\{\text{- definition of } flatten \text{ -}\} \\
&StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle
\end{aligned}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{run}_{\mathit{StateT}}\ (\mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda s_1 \to \mathit{StateT}\ \$\ \lambda s_2 \to \alpha^{-1}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2))\ s_1)\ s_2 \\
= & \quad \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ ((\lambda s_1 \to \mathit{StateT}\ \$\ \lambda s_2 \to \alpha^{-1}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2))\ s_1)\ s_2 \\
= & \quad \{\text{- function application -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda s_2 \to \alpha^{-1}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2))\ s_2 \\
= & \quad \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ (\lambda s_2 \to \alpha^{-1}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2))\ s_2 \\
= & \quad \{\text{- function application -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ (\alpha^{-1}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2)) \\
= & \quad \{\text{- definition of } \langle\$\rangle \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{fmap}\ \alpha\ (\mathit{fmap}\ \alpha^{-1}\ (\mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2))) \\
= & \quad \{\text{- Equation (2.2) -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{fmap}\ (\alpha \circ \alpha^{-1})\ (\mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2)) \\
= & \quad \{\text{- } \alpha \circ \alpha^{-1} = \mathit{id} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{fmap}\ \mathit{id}\ (\mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2)) \\
= & \quad \{\text{- Equation (2.1) -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ t\ (s_1, s_2) \\
= & \quad \{\text{- } \eta\text{-reduction -}\} \\
& \mathit{StateT}\ \$\ \mathit{run}_{\mathit{StateT}}\ t \\
= & \quad \{\text{- } \mathit{StateT} \circ \mathit{run}_{\mathit{StateT}} = \mathit{id} \text{ -}\} \\
& t
\end{aligned}
$$

Then, we first calculate the LHS $\mathit{flatten} \circ h_{\mathit{States}}$ into one function h'_{States} which is defined as

$$
\begin{aligned}
& h'_{\mathit{States}} :: \mathit{Functor}\ f \Rightarrow \mathit{Free}\ (\mathit{State}_F\ s_1\ :+:\ \mathit{State}_F\ s_2\ :+:\ f)\ a \to \mathit{StateT}\ (s_1, s_2)\ (\mathit{Free}\ f)\ a \\
& h'_{\mathit{States}}\ t = \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2
\end{aligned}
$$

For all $t :: \mathit{Free}\ (\mathit{State}_F\ s_1\ :+:\ \mathit{State}_F\ s_2\ :+:\ f)\ a$, we show the equation $(\mathit{flatten} \circ h_{\mathit{States}})\ t = h'_{\mathit{States}}\ t$ by the following calculation.

$$
\begin{aligned}
& (\mathit{flatten} \circ h_{\mathit{States}})\ t \\
= & \quad \{\text{- definition of } h_{\mathit{States}} \text{ -}\} \\
& (\mathit{flatten} \circ (\lambda t \to \mathit{StateT}\ (h_{\mathit{State}} \circ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t))))\ t \\
= & \quad \{\text{- function application -}\} \\
& \mathit{flatten}\ (\mathit{StateT}\ (h_{\mathit{State}} \circ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t))) \\
= & \quad \{\text{- definition of } \mathit{flatten} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (\mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ (h_{\mathit{State}} \circ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)))\ s_1)\ s_2 \\
= & \quad \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ ((h_{\mathit{State}} \circ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t))\ s_1)\ s_2 \\
= & \quad \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
& h'_{\mathit{States}}\ t
\end{aligned}
$$

Now we only need to show that for any input $t :: \mathit{Free}\ (\mathit{State}_F\ s_1\ :+:\ \mathit{State}_F\ s_2\ :+:\ f)\ a$, the equation $h'_{\mathit{States}}\ t = (h_{\mathit{State}} \circ \mathit{states2state})\ t$ holds. Note that both sides use folds. We can

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

proceed with either fold fusion, as what we have done in the proofs of other theorems, or a direct structural induction on the input t. Although using fold fusion makes the proof simpler than using structural induction, we opt for the latter here to show that the our methods of defining effects and translations based on algebraic effects and handlers also work well with structural induction.

case $t = Var\ x$

$$
\begin{aligned}
& (h_{State} \circ states2state)\ (Var\ x) \\
= \quad & \{\text{- definition of } states2state \text{ -}\} \\
& h_{State}\ (Var\ x) \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& StateT\ \$\ \lambda s \to \eta\ (x, s) \\
= \quad & \{\text{- let } s = (s_1, s_2) \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to Var\ (x, (s_1, s_2)) \\
= \quad & \{\text{- definition of } \alpha \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to Var\ (\alpha\ ((x, s_1), s_2)) \\
= \quad & \{\text{- definition of } fmap \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ Var\ ((x, s_1), s_2) \\
= \quad & \{\text{- definition of } \eta \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ \eta\ ((x, s_1), s_2) \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ (\lambda s \to \eta\ ((x, s_1), s))\ s_2 \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ run_{StateT}\ (StateT\ \$\ \lambda s \to \eta\ ((x, s_1), s))\ s_2 \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ run_{StateT}\ (h_{State}\ (\eta\ (x, s_1)))\ s_2 \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ run_{StateT}\ (h_{State}\ ((\lambda s \to \eta\ (x, s))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ run_{StateT}\ (h_{State}\ (run_{StateT}\ (StateT\ \$\ \lambda s \to \eta\ (x, s))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to fmap\ \alpha\ \$\ run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ (Var\ x))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } \langle\$\rangle \text{ -}\} \\
& StateT\ \$\ \lambda (s_1, s_2) \to \alpha\ \langle\$\rangle\ run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ (Var\ x))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h'_{States} \text{ -}\} \\
& h'_{States}\ (Var\ x)
\end{aligned}
$$

case $t = Op\ (Inl\ (Get\ k))$

Induction hypothesis: $h'_{States}\ (k\ s) = (h_{State} \circ states2state)\ (k\ s)$ for any s.

$$
\begin{aligned}
& (h_{State} \circ states2state)\ (Op\ (Inl\ (Get\ k))) \\
= \quad & \{\text{- definition of } states2state \text{ -}\} \\
& h_{State}\ \$\ get \gg\!= \lambda (s_1, \_) \to states2state\ (k\ s_1) \\
= \quad & \{\text{- definition of } get \text{ -}\} \\
& h_{State}\ \$\ Op\ (Inl\ (Get\ \eta)) \gg\!= \lambda (s_1, \_) \to states2state\ (k\ s_1) \\
= \quad & \{\text{- definition of } (\gg\!=) \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_{State}\ (Op\ (Inl\ (Get\ (\lambda(s_1, \_) \to states2state\ (k\ s_1)))))\\
=\ & \{\text{- definition of } h_{State} \text{ -}\}\\
& StateT\ \$\ \lambda s \to run_{StateT}\ ((\lambda(s_1, \_) \to h_{State}\ (states2state\ (k\ s_1)))\ s)\ s\\
=\ & \{\text{- let } s = (s_1, s_2) \text{ -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ ((\lambda(s_1, \_) \to h_{State}\ (states2state\ (k\ s_1)))\ (s_1, s_2))\ (s_1, s_2)\\
=\ & \{\text{- function application -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ (h_{State}\ (states2state\ (k\ s_1)))\ (s_1, s_2)\\
=\ & \{\text{- induction hypothesis -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ (h'_{States}\ (k\ s_1))\ (s_1, s_2)\\
=\ & \{\text{- definition of } h'_{States} \text{ -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ (StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\\
& \quad run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ (k\ s_1))\ s_1))\ s_2)\ (s_1, s_2)\\
=\ & \{\text{- definition of } run_{StateT} \text{ -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\\
& \quad run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ (k\ s_1))\ s_1))\ s_2)\ (s_1, s_2)\\
=\ & \{\text{- function application -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ (k\ s_1))\ s_1))\ s_2\\
=\ & \{\text{- } \beta\text{-expansion -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\\
& \quad run_{StateT}\ (h_{State}\ ((\lambda s \to run_{StateT}\ (h_{State}\ (k\ s))\ s)\ s_1))\ s_2\\
=\ & \{\text{- definition of } run_{StateT} \text{ -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\\
& \quad run_{StateT}\ (h_{State}\ (run_{StateT}\ (StateT\ \$\ \lambda s \to run_{StateT}\ (h_{State}\ (k\ s))\ s)\ s_1))\ s_2\\
=\ & \{\text{- definition of } h_{State} \text{ -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Get\ k))))\ s_1))\ s_2\\
=\ & \{\text{- definition of } h'_{States} \text{ -}\}\\
& h'_{States}\ (Op\ (Inl\ (Get\ k)))
\end{aligned}
$$

case $t = Op\ (Inl\ (Put\ s\ k))$

Induction hypothesis: $h'_{States}\ k = (h_{State} \circ states2state)\ k$.

$$
\begin{aligned}
& (h_{State} \circ states2state)\ (Op\ (Inl\ (Put\ s\ k)))\\
=\ & \{\text{- definition of } states2state \text{ -}\}\\
& h_{State}\ \$\ get \gg\!= \lambda(\_, s_2) \to put\ (s, s_2) \gg (states2state\ k)\\
=\ & \{\text{- definition of } get \text{ and } put \text{ -}\}\\
& h_{State}\ \$\ Op\ (Inl\ (Get\ \eta)) \gg\!= \lambda(\_, s_2) \to Op\ (Inl\ (Put\ (s, s_2)\ (\eta\ ()))) \gg (states2state\ k)\\
=\ & \{\text{- definition of } (\gg\!=) \text{ and } (\gg) \text{ -}\}\\
& h_{State}\ \$\ Op\ (Inl\ (Get\ (\lambda(\_, s_2) \to Op\ (Inl\ (Put\ (s, s_2)\ (states2state\ k))))))\\
=\ & \{\text{- definition of } h_{State} \text{ -}\}\\
& alg_S\ (Get\ (\lambda(\_, s_2) \to h_{State}\ (Op\ (Inl\ (Put\ (s, s_2)\ (states2state\ k))))))\\
=\ & \{\text{- definition of } alg_S \text{ -}\}\\
& StateT\ \$\ \lambda s' \to run_{StateT}\\
& \quad ((\lambda(\_, s_2) \to h_{State}\ (Op\ (Inl\ (Put\ (s, s_2)\ (states2state\ k)))))\ s')\ s'\\
=\ & \{\text{- let } s' = (s_1, s_2) \text{ -}\}\\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\\
& \quad ((\lambda(\_, s_2) \to h_{State}\ (Op\ (Inl\ (Put\ (s, s_2)\ (states2state\ k)))))\ (s_1, s_2))\ (s_1, s_2)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- function application -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}} \\
&\qquad (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inl}\ (\mathit{Put}\ (s, s_2)\ (\mathit{states2state}\ k))))))\ (s_1, s_2) \\
&= \quad \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}} \\
&\qquad (\mathit{StateT}\ \$\ \lambda\_ \to \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{states2state}\ k))\ (s, s_2))\ (s_1, s_2) \\
&= \quad \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to (\lambda\_ \to \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{states2state}\ k))\ (s, s_2))\ (s_1, s_2) \\
&= \quad \{\text{- function application -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{states2state}\ k))\ (s, s_2) \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ (h'_{\mathit{States}}\ k)\ (s, s_2) \\
&= \quad \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
&\qquad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ k)\ s_1))\ s_2)\ (s, s_2) \\
&= \quad \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
&\qquad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ k)\ s_1))\ s_2)\ (s, s_2) \\
&= \quad \{\text{- function application -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
&\qquad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ k)\ s))\ s_2 \\
&= \quad \{\text{- } \beta\text{-expansion -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
&\qquad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ ((\lambda s' \to \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ k)\ s)\ s_1))\ s_2 \\
&= \quad \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
&\qquad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda s' \to \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ k)\ s)\ s_1))\ s_2 \\
&= \quad \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
&\qquad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inl}\ (\mathit{Put}\ s\ k))))\ s_1))\ s_2 \\
&= \quad \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
&\quad h'_{\mathit{States}}\ (\mathit{Op}\ (\mathit{Inl}\ (\mathit{Put}\ s\ k)))
\end{aligned}
$$

case $t = \mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Get}\ k)))$

Induction hypothesis: $h'_{\mathit{States}}\ (k\ s) = (h_{\mathit{State}} \circ \mathit{states2state})\ (k\ s)$ for any s.

$$
\begin{aligned}
&\quad (h_{\mathit{State}} \circ \mathit{states2state})\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Get}\ k)))) \\
&= \quad \{\text{- definition of } \mathit{states2state} \text{ -}\} \\
&\quad h_{\mathit{State}}\ \$\ \mathit{get} \gg\!= \lambda(\_, s_2) \to \mathit{states2state}\ (k\ s_2) \\
&= \quad \{\text{- definition of } \mathit{get} \text{ -}\} \\
&\quad h_{\mathit{State}}\ \$\ \mathit{Op}\ (\mathit{Inl}\ (\mathit{Get}\ \eta)) \gg\!= \lambda(\_, s_2) \to \mathit{states2state}\ (k\ s_2) \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
&\quad h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inl}\ (\mathit{Get}\ (\lambda(\_, s_2) \to \mathit{states2state}\ (k\ s_2))))) \\
&= \quad \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
&\quad \mathit{StateT}\ \$\ \lambda s \to \mathit{run}_{\mathit{StateT}}\ ((\lambda(\_, s_2) \to h_{\mathit{State}}\ (\mathit{states2state}\ (k\ s_2)))\ s)\ s \\
&= \quad \{\text{- let } s = (s_1, s_2) \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ ((\lambda(\_, s_2) \to h_{\mathit{State}}\ (\mathit{states2state}\ (k\ s_2)))\ (s_1, s_2))\ (s_1, s_2) \\
= \quad & \{\text{- function application -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{states2state}\ (k\ s_2)))\ (s_1, s_2) \\
= \quad & \{\text{- induction hypothesis -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ (h'_{\mathit{States}}\ (k\ s_2))\ (s_1, s_2) \\
= \quad & \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (k\ s_2))\ s_1))\ s_2)\ (s_1, s_2) \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (k\ s_2))\ s_1))\ s_2)\ (s_1, s_2) \\
= \quad & \{\text{- function application -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (k\ s_2))\ s_1))\ s_2 \\
= \quad & \{\text{- reformulation -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad (\mathit{run}_{\mathit{StateT}}\ ((h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}} \circ k)\ s_2)\ s_2) \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad (\lambda s \to \mathit{run}_{\mathit{StateT}}\ ((h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}} \circ k)\ s)\ s)\ s_2 \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda s \to \mathit{run}_{\mathit{StateT}}\ ((h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}} \circ k)\ s)\ s)\ s_2 \\
= \quad & \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inl}\ (\mathit{Get}\ ((\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}} \circ k)))))\ s_2 \\
= \quad & \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1)\ (\mathit{Inl}\ (\mathit{Get}\ (h_{\mathit{State}} \circ k)))))\ s_2 \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ ((\lambda s \to \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s)\ (\mathit{Inl}\ (\mathit{Get}\ (h_{\mathit{State}} \circ k))))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}} \\
& \quad (\mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda s \to \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s)\ (\mathit{Inl}\ (\mathit{Get}\ (h_{\mathit{State}} \circ k))))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Get}\ k)))))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
& h'_{\mathit{States}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Get}\ k))))
\end{aligned}
$$

case $t = \mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Put}\ s\ k)))$

Induction hypothesis: $h'_{\mathit{States}}\ k = (h_{\mathit{State}} \circ \mathit{states2state})\ k$.

$$
\begin{aligned}
& (h_{\mathit{State}} \circ \mathit{states2state})\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Put}\ s\ k)))) \\
= \quad & \{\text{- definition of } \mathit{states2state} \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_{State}\ \$\ get \gg\!= \lambda(s_1, \_) \to put\ (s_1, s) \gg (states2state\ k) \\
= \quad & \{\text{- definition of } get \text{ and } put \text{ -}\} \\
& h_{State}\ \$\ Op\ (Inl\ (Get\ \eta)) \gg\!= \lambda(s_1, \_) \to Op\ (Inl\ (Put\ (s_1, s)\ (\eta\ ()))) \gg (states2state\ k) \\
= \quad & \{\text{- definition of } (\gg\!=) \text{ and } (\gg) \text{ -}\} \\
& h_{State}\ \$\ Op\ (Inl\ (Get\ (\lambda(s_1, \_) \to Op\ (Inl\ (Put\ (s_1, s)\ (states2state\ k))))))) \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& StateT\ \$\ \lambda s' \to run_{StateT} \\
& \quad ((\lambda(s_1, \_) \to h_{State}\ (Op\ (Inl\ (Put\ (s_1, s)\ (states2state\ k))))))\ s')\ s' \\
= \quad & \{\text{- let } s' = (s_1, s_2) \text{ -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT} \\
& \quad ((\lambda(s_1, \_) \to h_{State}\ (Op\ (Inl\ (Put\ (s_1, s)\ (states2state\ k))))))\ (s_1, s_2))\ (s_1, s_2) \\
= \quad & \{\text{- function application -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT} \\
& \quad (h_{State}\ (Op\ (Inl\ (Put\ (s_1, s)\ (states2state\ k))))))\ (s_1, s_2) \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ (StateT\ \$ \\
& \quad \lambda s' \to run_{StateT}\ (h_{State}\ (states2state\ k))\ (s_1, s))\ (s_1, s_2) \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to (\lambda s' \to run_{StateT}\ (h_{State}\ (states2state\ k))\ (s_1, s))\ (s_1, s_2) \\
= \quad & \{\text{- function application -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ (h_{State}\ (states2state\ k))\ (s_1, s) \\
= \quad & \{\text{- induction hypothesis -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ (h'_{States}\ k)\ (s_1, s) \\
= \quad & \{\text{- definition of } h'_{States} \text{ -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to run_{StateT}\ (StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ k)\ s_1))\ s_2)\ (s_1, s) \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ k)\ s_1))\ s_2)\ (s_1, s) \\
= \quad & \{\text{- function application -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ k)\ s_1))\ s \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad (\lambda s' \to run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ k)\ s_1))\ s)\ s_2 \\
= \quad & \{\text{- definition of } run_{StateT} \text{ -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad run_{StateT}\ (StateT\ \$\ \lambda s' \to run_{StateT}\ (h_{State}\ (run_{StateT}\ (h_{State}\ k)\ s_1))\ s)\ s_2 \\
= \quad & \{\text{- definition of } h_{State} \text{ -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ s\ (run_{StateT}\ (h_{State}\ k)\ s_1)))))\ s_2 \\
= \quad & \{\text{- reformulation -}\} \\
& StateT\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad run_{StateT}\ (h_{State}\ (Op\ (Inl\ (Put\ s\ ((\lambda k \to run_{StateT}\ k\ s_1)\ (h_{State}\ k))))))\ s_2 \\
= \quad & \{\text{- definition of } fmap \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1)\ (\mathit{Inl}\ (\mathit{Put}\ s\ (h_{\mathit{State}}\ k)))))\ s_2 \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}} \\
& \quad (h_{\mathit{State}}\ ((\lambda s' \to \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s')\ (\mathit{Inl}\ (\mathit{Put}\ s\ (h_{\mathit{State}}\ k))))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$ \\
& \quad \lambda s' \to \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s')\ (\mathit{Inl}\ (\mathit{Put}\ s\ (h_{\mathit{State}}\ k))))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Put}\ s\ k)))))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
& h'_{\mathit{States}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Put}\ s\ k))))
\end{aligned}
$$

case $t = \mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ y))$

Induction hypothesis: $h'_{\mathit{States}}\ y = (h_{\mathit{State}} \circ \mathit{states2state})\ y$.

$$
\begin{aligned}
& (h_{\mathit{State}} \circ \mathit{states2state})\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ y))) \\
= \quad & \{\text{- definition of } \mathit{states2state} \text{ -}\} \\
& h_{\mathit{State}}\ \$\ \mathit{Op}\ (\mathit{Inr}\ (\mathit{fmap}\ \mathit{states2state}\ y)) \\
= \quad & \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
& \mathit{fwd}_S\ (\mathit{fmap}\ (h_{\mathit{State}} \circ \mathit{states2state})\ y) \\
= \quad & \{\text{- induction hypothesis -}\} \\
& \mathit{fwd}_S\ (\mathit{fmap}\ h'_{\mathit{States}}\ y) \\
= \quad & \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
& \mathit{fwd}_S\ (\mathit{fmap}\ (\lambda t \to \mathit{StateT} \\
& \quad \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ y) \\
= \quad & \{\text{- definition of } \mathit{fwd}_S \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda s \to \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s)\ (\mathit{fmap}\ (\lambda t \to \mathit{StateT} \\
& \quad \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ y) \\
= \quad & \{\text{- Equation (2.2) -}\} \\
& \mathit{StateT}\ \$\ \lambda s \to \mathit{Op}\ (\mathit{fmap}\ ((\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s) \circ (\lambda t \to \mathit{StateT} \\
& \quad \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2))\ y) \\
= \quad & \{\text{- reformulation -}\} \\
& \mathit{StateT}\ \$\ \lambda s \to \mathit{Op}\ (\mathit{fmap}\ (\lambda t \to \mathit{run}_{\mathit{StateT}}\ (\mathit{StateT} \\
& \quad \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ s)\ y) \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda s \to \mathit{Op}\ (\mathit{fmap}\ (\lambda t \to \\
& \quad (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ s)\ y) \\
= \quad & \{\text{- let } s = (s_1, s_2) \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{Op}\ (\mathit{fmap}\ (\lambda t \to \\
& \quad (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ (s_1, s_2))\ y) \\
= \quad & \{\text{- function application -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{Op}\ (\mathit{fmap}\ (\lambda t \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ y) \\
= \quad & \{\text{- reformulation -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \mathit{Op}\ (\mathit{fmap}\ (\alpha\ \langle\$\rangle \\
& \quad \circ\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_2) \circ h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}})\ y) \\
= \quad & \{\text{- definition of } \langle\$\rangle \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{Op}\ (\mathit{fmap}\ ((\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_2) \circ h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}})\ y) \\
= \quad & \{\text{- Equation (2.2) -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_2)\ (\mathit{fmap}\ (h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}})\ y) \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ (\lambda s \to \mathit{Op}\ \$ \\
& \quad \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s)\ (\mathit{fmap}\ (h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}})\ y))\ s_2 \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda s \to \mathit{Op}\ \$ \\
& \quad \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s)\ (\mathit{fmap}\ (h_{\mathit{State}} \circ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}})\ y))\ s_2 \\
= \quad & \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}} \\
& \quad (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{fmap}\ ((\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1) \circ h_{\mathit{State}})\ y))))\ s_2 \\
= \quad & \{\text{- Equation (2.2) -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}} \\
& \quad (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1)\ (\mathit{fmap}\ h_{\mathit{State}}\ y)))))\ s_2 \\
= \quad & \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}} \\
& \quad (h_{\mathit{State}}\ (\mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s_1)\ (\mathit{Inr}\ (\mathit{fmap}\ h_{\mathit{State}}\ y))))\ s_2 \\
= \quad & \{\text{- } \beta\text{-expansion -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}} \\
& \quad ((\lambda s \to \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s)\ (\mathit{Inr}\ (\mathit{fmap}\ h_{\mathit{State}}\ y)))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$ \\
& \quad \lambda s \to \mathit{Op}\ \$\ \mathit{fmap}\ (\lambda k \to \mathit{run}_{\mathit{StateT}}\ k\ s)\ (\mathit{Inr}\ (\mathit{fmap}\ h_{\mathit{State}}\ y)))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h_{\mathit{State}} \text{ -}\} \\
& \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ y))))\ s_1))\ s_2 \\
= \quad & \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
& h'_{\mathit{States}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ y)))
\end{aligned}
$$

■

5 Proofs for the all in one simulation

In this section, we prove the correctness of the final simulation in Section 6.2.

Theorem 5. $\mathit{simulate} = h_{\mathit{Local}}$

Proof We calculate as follows, using all our three previous theorems Theorem 1, Theorem 3, Theorem 4, and an auxiliary lemma Lemma 15.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{simulate} \\
= \quad & \{\text{- definition of } \mathit{simulate} \text{ -}\} \\
& \mathit{extract} \circ h_{\mathit{State}} \circ \mathit{states2state} \circ \mathit{nondet2state} \circ (\Leftrightarrow) \circ \mathit{local2global} \\
= \quad & \{\text{- Theorem 4 -}\} \\
& \mathit{extract} \circ \mathit{flatten} \circ h_{\mathit{States}} \circ \mathit{nondet2state} \circ (\Leftrightarrow) \circ \mathit{local2global} \\
= \quad & \{\text{- Lemma 15 -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}} \circ \mathit{extract}_{\mathit{SS}} \circ h_{\mathit{State}} \circ \mathit{nondet2state} \circ (\Leftrightarrow) \circ \mathit{local2global} \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{ND+f}} \text{ -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}} \circ \mathit{run}_{\mathit{ND+f}} \circ (\Leftrightarrow) \circ \mathit{local2global} \\
= \quad & \{\text{- Theorem 3 -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow) \circ \mathit{local2global} \\
= \quad & \{\text{- definition of } h_{\mathit{Global}} \text{ -}\} \\
& h_{\mathit{Global}} \circ \mathit{local2global} \\
= \quad & \{\text{- Theorem 1 -}\} \\
& h_{\mathit{Local}}
\end{aligned}
$$

■

Lemma 15. $\mathit{extract} \circ \mathit{flatten} \circ h_{\mathit{States}} = \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}} \circ \mathit{extract}_{\mathit{SS}} \circ h_{\mathit{State}}$

Proof As shown in Appendix 4, we can combine $\mathit{flatten} \circ h_{\mathit{States}}$ into one function h'_{States} defined as follows:

$$
\begin{aligned}
& h'_{\mathit{States}} :: \mathit{Functor}\ f \Rightarrow \mathit{Free}\ (\mathit{State}_F\ s_1 :+: \mathit{State}_F\ s_2 :+: f)\ a \to \mathit{StateT}\ (s_1, s_2)\ (\mathit{Free}\ f)\ a \\
& h'_{\mathit{States}}\ t = \mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2
\end{aligned}
$$

Then we show that for any input $t :: \mathit{Free}\ (\mathit{State}_F\ (\mathit{SS}\ (\mathit{State}_F\ s :+: f)\ a) :+: (\mathit{State}_F\ s :+: f))\ ()$, we have $(\mathit{extract} \circ h'_{\mathit{States}})\ t = (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}} \circ \mathit{extract}_{\mathit{SS}} \circ h_{\mathit{State}})\ t$ via the following calculation.

$$
\begin{aligned}
& (\mathit{extract} \circ h'_{\mathit{States}})\ t \\
= \quad & \{\text{- function application -}\} \\
& \mathit{extract}\ (h'_{\mathit{States}}\ t) \\
= \quad & \{\text{- definition of } h'_{\mathit{States}} \text{ -}\} \\
& \mathit{extract}\ (\mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2) \\
= \quad & \{\text{- definition of } \mathit{extract} \text{ -}\} \\
& \lambda s \to \mathit{results}_{\mathit{SS}} \circ \mathit{fst} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (\mathit{StateT}\ \$\ \lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ (\mathit{SS}\ [\,]\ [\,], s) \\
= \quad & \{\text{- definition of } \mathit{run}_{\mathit{StateT}} \text{ -}\} \\
& \lambda s \to \mathit{results}_{\mathit{SS}} \circ \mathit{fst} \circ \mathit{snd}\ \langle\$\rangle\ (\lambda(s_1, s_2) \to \alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ s_1))\ s_2)\ (\mathit{SS}\ [\,]\ [\,], s) \\
= \quad & \{\text{- function application -}\} \\
& \lambda s \to \mathit{results}_{\mathit{SS}} \circ \mathit{fst} \circ \mathit{snd}\ \langle\$\rangle\ (\alpha\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,])))\ s) \\
= \quad & \{\text{- Equation (2.2) -}\} \\
& \lambda s \to \mathit{results}_{\mathit{SS}} \circ \mathit{fst} \circ \mathit{snd} \circ \alpha\ \langle\$\rangle
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,])))\ s \\
= \quad & \{\text{- } \mathit{fst} \circ \mathit{snd} \circ \alpha = \mathit{snd} \circ \mathit{fst} \text{ -}\} \\
& \lambda s \to \mathit{results}_{\mathit{SS}} \circ \mathit{snd} \circ \mathit{fst}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,])))\ s \\
= \quad & \{\text{- Equation (2.2) and definition of } \langle\$\rangle \text{ -}\} \\
& \lambda s \to \mathit{fmap}\ (\mathit{results}_{\mathit{SS}} \circ \mathit{snd}) \circ \mathit{fmap}\ \mathit{fst}\ \$ \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,])))\ s \\
= \quad & \{\text{- definition of } \mathit{flip} \text{ and reformulation -}\} \\
& \lambda s \to \mathit{fmap}\ (\mathit{results}_{\mathit{SS}} \circ \mathit{snd}) \circ \mathit{fmap}\ \mathit{fst}\ \$ \\
& \quad \mathit{flip}\ (\mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}})\ s\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,])) \\
= \quad & \{\text{- reformulation -}\} \\
& \lambda s \to \mathit{fmap}\ (\mathit{results}_{\mathit{SS}} \circ \mathit{snd}) \circ (\mathit{fmap}\ \mathit{fst} \circ \mathit{flip}\ (\mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}})\ s)\ \$ \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,]) \\
= \quad & \{\text{- parametricity of free monads -}\} \\
& \lambda s \to (\mathit{fmap}\ \mathit{fst} \circ \mathit{flip}\ (\mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}})\ s) \circ \mathit{fmap}\ (\mathit{results}_{\mathit{SS}} \circ \mathit{snd})\ \$ \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,]) \\
= \quad & \{\text{- definition of } \langle\$\rangle \text{ -}\} \\
& \lambda s \to (\mathit{fmap}\ \mathit{fst} \circ \mathit{flip}\ (\mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}})\ s)\ \$\ \mathit{results}_{\mathit{SS}} \circ \mathit{snd}\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,]) \\
= \quad & \{\text{- function application -}\} \\
& \lambda s \to \mathit{fmap}\ \mathit{fst}\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{results}_{\mathit{SS}} \circ \mathit{snd}\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,])))\ s) \\
= \quad & \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
& \lambda s \to \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ (\mathit{results}_{\mathit{SS}} \circ \mathit{snd}\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,])))\ s \\
= \quad & \{\text{- reformulation -}\} \\
& \lambda s \to (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}}\ \$\ \mathit{results}_{\mathit{SS}} \circ \mathit{snd}\ \langle\$\rangle \\
& \quad \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,]))\ s \\
= \quad & \{\text{- } \eta\text{-reduction -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}}\ \$\ \mathit{results}_{\mathit{SS}} \circ \mathit{snd}\ \langle\$\rangle\ \mathit{run}_{\mathit{StateT}}\ (h_{\mathit{State}}\ t)\ (\mathit{SS}\ [\,]\ [\,]\ [\,]) \\
= \quad & \{\text{- definition of } \mathit{extract}_{\mathit{SS}} \text{ -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}}\ \$\ \mathit{extract}_{\mathit{SS}}\ (h_{\mathit{State}}\ t) \\
= \quad & \{\text{- -}\} \\
& (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ \mathit{run}_{\mathit{StateT}} \circ h_{\mathit{State}} \circ \mathit{extract}_{\mathit{SS}} \circ h_{\mathit{State}})\ t
\end{aligned}
$$

Note that in the above calculation, we use the parametricity (Reynolds, 1983; Wadler, 1989) of free monads which is formally stated as follows:

$$\mathit{fmap}\ f \circ g = g \circ \mathit{fmap}\ f$$

for any $g :: \forall a.\mathit{Free}\ F\ a \to \mathit{Free}\ G\ a$ with two functors F and G.

■

6 Proofs for modelling local state with undo

In this section, we prove the following theorem in Section 7.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Theorem 6. *Given Functor f and Undo s u, the equation*

$$h_{GlobalM} \circ local2global_M = h_{LocalM}$$

holds for all programs $p :: Free\,(Modify_F\ s\ u :+: Nondet_F :+: f)\ a$ *that do not use the operation* $Op\,(Inl\ MRestore\ \_\ \_)$.

The proof structure is very similar to that in Appendix 2. We start with the following preliminary fusion.

Preliminary. It is easy to see that $run_{StateT} \circ h_{Modify}$ can be fused into a single fold defined as follows:

$$\begin{aligned}
&h_{Modify1} :: (Functor\ f, Undo\ s\ r) \Rightarrow Free\,(Modify_F\ s\ r :+: f)\ a \to (s \to Free\ f\ (a, s))\\
&h_{Modify1} = fold\ gen_S\ (alg_S \,\#\, fwd_S)\\
&\quad \textbf{where}\\
&\quad\quad gen_S\ x \qquad\qquad s = Var\,(x, s)\\
&\quad\quad alg_S\,(MGet\ k) \qquad s = k\ s\ s\\
&\quad\quad alg_S\,(MUpdate\ r\ k)\ s = k\,(s \oplus r)\\
&\quad\quad alg_S\,(MRestore\ r\ k)\ s = k\,(s \ominus r)\\
&\quad\quad fwd_S\ y \qquad\qquad s = Op\,(fmap\ (\$s)\ y)
\end{aligned}$$

For brevity, we use $h_{Modify1}$ to replace $run_{StateT} \circ h_{Modify}$ in the following proofs.

6.1 Main proof structure

The main proof structure of Theorem 6 is as follows.

Proof Both the left-hand side and the right-hand side of the equation consist of function compositions involving one or more folds. We apply fold fusion separately on both sides to contract each into a single fold:

$$\begin{aligned}
h_{GlobalM} \circ local2global_M &= fold\ gen_{LHS}\ (alg^S_{LHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{LHS})\\
h_{LocalM} &= fold\ gen_{RHS}\ (alg^S_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS})
\end{aligned}$$

Finally, we show that both folds are equal by showing that their corresponding parameters are equal:

$$\begin{aligned}
gen_{LHS} &= gen_{RHS}\\
alg^S_{LHS} &= alg^S_{RHS}\\
alg^{ND}_{LHS} &= alg^{ND}_{RHS}\\
fwd_{LHS} &= fwd_{RHS}
\end{aligned}$$

We elaborate each of these steps below. ■

6.2 Fusing the right-hand side

We calculate as follows:

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_{LocalM} \\
= \quad & \{\text{- definition -}\} \\
& h_L \circ h_{Modify1} \\
& \quad \text{with} \\
& \quad\quad h_L :: (Functor\, f, Functor\, g) \\
& \quad\quad\quad \Rightarrow g\,(Free\,(Nondet_F :+: f)\,(a, s)) \to g\,(Free\, f\,[a]) \\
& \quad\quad h_L = fmap\,(fmap\,(fmap\, fst) \circ h_{ND+f}) \\
= \quad & \{\text{- definition of } h_{Modify1} \text{ -}\} \\
& h_L \circ fold\, gen_S\,(alg_S \,\#\, fwd_S) \\
= \quad & \{\text{- fold fusion-post (Equation 3.2) -}\} \\
& fold\, gen_{RHS}\,(alg^S_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS})
\end{aligned}
$$

This last step is valid provided that the fusion conditions are satisfied:

$$
\begin{aligned}
h_L \circ gen_S &= gen_{RHS} && (1) \\
h_L \circ (alg_S \,\#\, fwd_S) &= (alg^S_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS}) \circ fmap\, h_L && (2)
\end{aligned}
$$

For the first fusion condition (1), we define gen_{RHS} as follows

$$
\begin{aligned}
& gen_{RHS} :: Functor\, f \Rightarrow a \to (s \to Free\, f\,[a]) \\
& gen_{RHS}\, x = \lambda s \to Var\,[x]
\end{aligned}
$$

We show that (1) is satisfied by the following calculation.

$$
\begin{aligned}
& h_L\,(gen_S\, x) \\
= \quad & \{\text{- definition of } gen_S \text{ -}\} \\
& h_L\,(\lambda s \to Var\,(x, s)) \\
= \quad & \{\text{- definition of } h_L \text{ -}\} \\
& fmap\,(fmap\,(fmap\, fst) \circ h_{ND+f})\,(\lambda s \to Var\,(x, s)) \\
= \quad & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\,(fmap\, fst)\,(h_{ND+f}\,(Var\,(x, s))) \\
= \quad & \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\,(fmap\, fst)\,(Var\,[(x, s)]) \\
= \quad & \{\text{- definition of } fmap \text{ (twice) -}\} \\
& \lambda s \to Var\,[x] \\
= \quad & \{\text{- definition of } gen_{RHS} \text{ -}\} \\
= \quad & gen_{RHS}\, x
\end{aligned}
$$

By a straightforward case analysis on the two cases *Inl* and *Inr*, the second fusion condition (2) decomposes into two separate conditions:

$$
\begin{aligned}
h_L \circ alg_S &= alg^S_{RHS} \circ fmap\, h_L && (3) \\
h_L \circ fwd_S \circ Inl &= alg^{ND}_{RHS} \circ fmap\, h_L && (4) \\
h_L \circ fwd_S \circ Inr &= fwd_{RHS} \circ fmap\, h_L && (5)
\end{aligned}
$$

For the subcondition (3), we define alg^S_{RHS} as follows.

$$
\begin{aligned}
& alg^S_{RHS} :: Undo\, s\, r \Rightarrow Modify_F\, s\, r\,(s \to p) \to (s \to p) \\
& alg^S_{RHS}\,(MGet\, k) = \lambda s \to k\, s\, s \\
& alg^S_{RHS}\,(MUpdate\, r\, k) = \lambda s \to k\,(s \oplus r) \\
& alg^S_{RHS}\,(MRestore\, r\, k) = \lambda s \to k\,(s \oplus r)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

We prove its correctness by a case analysis on the shape of input $t :: State_F\ s\ (s \to Free\ (Nondet_F :+: f)\ (a, s))$.

case $t = Get\ k$

$$
\begin{aligned}
& h_L\ (alg_S\ (Get\ k)) \\
= \ & \{\text{- definition of } alg_S \text{ -}\} \\
& h_L\ (\lambda s \to k\ s\ s) \\
= \ & \{\text{- definition of } h_L \text{ -}\} \\
& fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to k\ s\ s) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ s\ s)) \\
= \ & \{\text{- beta-expansion (twice) -}\} \\
= \ & \lambda s \to (\lambda s_1\ s_2 \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ s_2\ s_1)))\ s\ s \\
= \ & \{\text{- definition of } fmap \text{ (twice) -}\} \\
= \ & \lambda s \to (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (\lambda s_1\ s_2 \to k\ s_2\ s_1))\ s\ s \\
= \ & \{\text{- eta-expansion of } k \text{ -}\} \\
= \ & \lambda s \to (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ k)\ s\ s \\
= \ & \{\text{- definition of } algRHS \text{ -}\} \\
= \ & alg^S_{RHS}\ (Get\ (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ k)) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
= \ & alg^S_{RHS}\ (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (Get\ k)) \\
= \ & \{\text{- definition of } h_L \text{ -}\} \\
= \ & alg^S_{RHS}\ (fmap\ h_L\ (Get\ k))
\end{aligned}
$$

case $t = MUpdate\ r\ k$

$$
\begin{aligned}
& h_L\ (alg_S\ (MUpdate\ r\ k)) \\
= \ & \{\text{- definition of } alg_S \text{ -}\} \\
& h_L\ (\lambda s \to k\ (s \oplus r)) \\
= \ & \{\text{- definition of } h_L \text{ -}\} \\
& fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to k\ (s \oplus r)) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ (s \oplus r))) \\
= \ & \{\text{- beta-expansion -}\} \\
= \ & \lambda s \to (\lambda s_1 \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ s_1)))\ (s \oplus r) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
= \ & \lambda s \to (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s_1 \to k\ s_1))\ (s \oplus r) \\
= \ & \{\text{- eta-expansion of } k \text{ -}\} \\
= \ & \lambda s \to (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ k)\ (s \oplus r) \\
= \ & \{\text{- definition of } alg^S_{RHS} \text{ -}\} \\
= \ & alg^S_{RHS}\ (MUpdate\ r\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ k)) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
= \ & alg^S_{RHS}\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (MUpdate\ r\ k)) \\
= \ & \{\text{- definition of } h_L \text{ -}\} \\
= \ & alg^S_{RHS}\ (fmap\ h_L\ (MUpdate\ r\ k))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

case $t = MRestore\ r\ k$

$$
\begin{aligned}
& h_L\ (alg_S\ (MRestore\ r\ k)) \\
= \ & \{\text{- definition of } alg_S \text{ -}\} \\
& h_L\ (\lambda s \to k\ (s \ominus r)) \\
= \ & \{\text{- definition of } h_L \text{ -}\} \\
& fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to k\ (s \ominus r)) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ (s \ominus r))) \\
= \ & \{\text{- beta-expansion -}\} \\
= \ & \lambda s \to (\lambda s_1 \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (k\ s_1)))\ (s \ominus r) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
= \ & \lambda s \to (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s_1 \to k\ s_1))\ (s \ominus r) \\
= \ & \{\text{- eta-expansion of } k \text{ -}\} \\
= \ & \lambda s \to (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ k)\ (s \ominus r) \\
= \ & \{\text{- definition of } alg^S_{RHS} \text{ -}\} \\
= \ & alg^S_{RHS}\ (MRestore\ r\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ k)) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
= \ & alg^S_{RHS}\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (MRestore\ r\ k)) \\
= \ & \{\text{- definition of } h_L \text{ -}\} \\
= \ & alg^S_{RHS}\ (fmap\ h_L\ (MRestore\ r\ k))
\end{aligned}
$$

For the subcondition (4), we define alg^{ND}_{RHS} as follows.

$$
\begin{aligned}
& alg^{ND}_{RHS} :: Functor\ f \Rightarrow Nondet_F\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a]) \\
& alg^{ND}_{RHS}\ Fail \quad\ = \lambda s \to Var\ [\,] \\
& alg^{ND}_{RHS}\ (Or\ p\ q) = \lambda s \to liftM2\ (+\!+)\ (p\ s)\ (q\ s)
\end{aligned}
$$

To show its correctness, given $op :: Nondet_F\ (s \to Free\ (Nondet_F :+: f)\ (a, s))$ with $Functor\ f$, we calculate:

$$
\begin{aligned}
& h_L\ (fwd_S\ (Inl\ op)) \\
= \ & \{\text{- definition of } fwd_S \text{ -}\} \\
& h_L\ (\lambda s \to Op\ (fmap\ (\$s)\ (Inl\ op))) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& h_L\ (\lambda s \to Op\ (Inl\ (fmap\ (\$s)\ op))) \\
= \ & \{\text{- definition of } h_L \text{ -}\} \\
& fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to Op\ (Inl\ (fmap\ (\$s)\ op))) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (Op\ (Inl\ (fmap\ (\$s)\ op)))) \\
= \ & \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (alg_{ND+f}\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ op)))
\end{aligned}
$$

We proceed by a case analysis on op:

case $op = Fail$

$$
\begin{aligned}
& \lambda s \to fmap\ (fmap\ fst)\ (alg_{ND+f}\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ Fail))) \\
= \ & \{\text{- defintion of } fmap \text{ (twice) -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \lambda s \to fmap\ (fmap\ fst)\ (alg_{ND+f}\ Fail) \\
= & \quad \{\text{- definition of } alg_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (Var\ [\,]) \\
= & \quad \{\text{- definition of } fmap \text{ (twice) -}\} \\
& \lambda s \to Var\ [\,] \\
= & \quad \{\text{- definition of } alg^{ND}_{RHS} \text{ -}\} \\
& alg^{ND}_{RHS}\ Fail \\
= & \quad \{\text{-definition of } fmap \text{ -}\} \\
& alg^{ND}_{RHS}\ (fmap\ h_L\ fail)
\end{aligned}
$$

case $op = Or\ p\ q$

$$
\begin{aligned}
& \lambda s \to fmap\ (fmap\ fst)\ (alg_{ND+f}\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ (Or\ p\ q)))) \\
= & \quad \{\text{- defintion of } fmap \text{ (twice) -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (alg_{ND+f}\ (Or\ (h_{ND+f}\ (p\ s))\ (h_{ND+f}\ (q\ s)))) \\
= & \quad \{\text{- definition of } alg_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (liftM2\ (+\!\!+)\ (h_{ND+f}\ (p\ s))\ (h_{ND+f}\ (q\ s))) \\
= & \quad \{\text{- Lemma 3 -}\} \\
& \lambda s \to liftM2\ (+\!\!+)\ (fmap\ (fmap\ fst)\ (h_{ND+f}\ (p\ s)))\ (fmap\ (fmap\ fst)\ (h_{ND+f}\ (q\ s))) \\
= & \quad \{\text{- definition of } alg^{ND}_{RHS} \text{ -}\} \\
& alg^{ND}_{RHS}\ (Or\ (fmap\ (fmap\ fst) \circ h_{ND+f} \circ p)\ (fmap\ (fmap\ fst) \circ h_{ND+f} \circ q)) \\
= & \quad \{\text{- defintion of } fmap \text{ (twice) -}\} \\
& alg^{ND}_{RHS}\ (fmap\ (fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f}))\ (Or\ p\ q)) \\
= & \quad \{\text{- defintion of } h_L \text{ -}\} \\
& alg^{ND}_{RHS}\ (fmap\ h_L\ (Or\ p\ q))
\end{aligned}
$$

For the last subcondition (5), we define fwd_{RHS} as follows.

$$
\begin{aligned}
& fwd_{RHS} :: Functor\ f \Rightarrow f\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a]) \\
& fwd_{RHS}\ op = \lambda s \to Op\ (fmap\ (\$s)\ op)
\end{aligned}
$$

To show its correctness, given input $op :: f\ (s \to Free\ (Nondet_F\ :+:\ f)\ (a, s))$, we calculate:

$$
\begin{aligned}
& h_L\ (fwd_S\ (Inr\ op)) \\
= & \quad \{\text{- definition of } fwd_S \text{ -}\} \\
& h_L\ (\lambda s \to Op\ (fmap\ (\$s)\ (Inr\ op))) \\
= & \quad \{\text{- definition of } fmap \text{ -}\} \\
& h_L\ (\lambda s \to Op\ (Inr\ (fmap\ (\$s)\ op))) \\
= & \quad \{\text{- definition of } h_L \text{ -}\} \\
& fmap\ (fmap\ (fmap\ fst) \circ h_{ND+f})\ (\lambda s \to Op\ (Inr\ (fmap\ (\$s)\ op))) \\
= & \quad \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (h_{ND+f}\ (Op\ (Inr\ (fmap\ (\$s)\ op)))) \\
= & \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (fwd_{ND+f}\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ op))) \\
= & \quad \{\text{- definition of } fwd_{ND+f} \text{ -}\} \\
& \lambda s \to fmap\ (fmap\ fst)\ (Op\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ op))) \\
= & \quad \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to Op\ (fmap\ (fmap\ (fmap\ fst))\ (fmap\ h_{ND+f}\ (fmap\ (\$s)\ op)))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{array}{ll}
= & \{\text{- } \mathit{fmap} \text{ fusion -}\} \\
& \lambda s \to \mathit{Op}\ (\mathit{fmap}\ (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ h_{ND+f})\ (\mathit{fmap}\ (\$s)\ \mathit{op})) \\
= & \{\text{- definition of } h_L \text{ -}\} \quad | \\
& \lambda s \to \mathit{Op}\ (h_L\ (\mathit{fmap}\ (\$s)\ \mathit{op})) \\
= & \{\text{- Lemma 2 -}\} \\
& \lambda s \to \mathit{Op}\ (\mathit{fmap}\ ((\$s) \circ h_L)\ \mathit{op}) \\
= & \{\text{- } \mathit{fmap} \text{ fusion -}\} \\
& \lambda s \to \mathit{Op}\ (\mathit{fmap}\ (\$s)\ (\mathit{fmap}\ h_L\ \mathit{op})) \\
= & \{\text{- definition of } \mathit{fwd}_{RHS} \text{ -}\} \\
& \mathit{fwd}_{RHS}\ (\mathit{fmap}\ h_L\ \mathit{op})
\end{array}
$$

6.3 Fusing the left-hand side

We calculate as follows:

$$
\begin{array}{ll}
& h_{GlobalM} \circ \mathit{local2global}_M \\
= & \{\text{- definition of } \mathit{local2global}_M \text{ -}\} \\
& h_{GlobalM} \circ \mathit{fold}\ \mathit{Var}\ \mathit{alg} \\
& \quad \textbf{where} \\
& \quad\quad \mathit{alg}\ (\mathit{Inl}\ (\mathit{MUpdate}\ r\ k)) = (\mathit{update}\ r \parallel \mathit{side}\ (\mathit{restore}\ r)) \gg\!= k \\
& \quad\quad \mathit{alg}\ p \qquad\qquad\qquad\quad = \mathit{Op}\ p \\
= & \{\text{- fold fusion-post' (Equation 3.3) -}\} \\
& \mathit{fold}\ \mathit{gen}_{LHS}\ (\mathit{alg}^S_{LHS} \# \mathit{alg}^{ND}_{LHS} \# \mathit{fwd}_{LHS})
\end{array}
$$

This last step is valid provided that the fusion conditions are satisfied:

$$h_{GlobalM} \circ \mathit{Var} \quad = \quad \mathit{gen}_{LHS} \qquad (1)$$

$$
\begin{array}{ll}
& h_{GlobalM} \circ \mathit{alg} \circ \mathit{fmap}\ \mathit{local2global}_M \\
= & (\mathit{alg}^S_{LHS} \# \mathit{alg}^{ND}_{LHS} \# \mathit{fwd}_{LHS}) \circ \mathit{fmap}\ h_{GlobalM} \circ \mathit{fmap}\ \mathit{local2global}_M \qquad (2)
\end{array}
$$

The first subcondition (1) is met by

$$
\begin{array}{l}
\mathit{gen}_{LHS} :: \mathit{Functor}\ f \Rightarrow a \to (s \to \mathit{Free}\ f\ [a]) \\
\mathit{gen}_{LHS}\ x = \lambda s \to \mathit{Var}\ [x]
\end{array}
$$

as established in the following calculation:

$$
\begin{array}{ll}
& h_{GlobalM}\ (\mathit{Var}\ x) \\
= & \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (\mathit{Var}\ x)))) \\
= & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{Modify1}\ (h_{ND+f}\ (\mathit{Var}\ x))) \\
= & \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{Modify1}\ (\mathit{Var}\ [x])) \\
= & \{\text{- definition of } h_{Modify1} \text{ -}\} \\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s \to \mathit{Var}\ ([x], s)) \\
= & \{\text{- definition of } \mathit{fmap} \text{ (twice) -}\}
\end{array}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \lambda s \to Var\,[x] \\
= \;& \{\text{- definition of } gen_{LHS} \text{ -}\} \\
& gen_{LHS}\,x
\end{aligned}
$$

We can split the second fusion condition (2) in three subconditions:

$$
\begin{aligned}
h_{GlobalM} \circ alg \circ Inl \circ fmap\ local2global_M &= alg^{S}_{LHS} \circ fmap\ h_{GlobalM} \circ fmap\ local2global_M && (3)\\
h_{GlobalM} \circ alg \circ Inr \circ Inl \circ fmap\ local2global_M &= alg^{ND}_{LHS} \circ fmap\ h_{GlobalM} \circ fmap\ local2global_M && (4)\\
h_{GlobalM} \circ alg \circ Inr \circ Inr \circ fmap\ local2global_M &= fwd_{LHS} \circ fmap\ h_{GlobalM} \circ fmap\ local2global_M && (5)
\end{aligned}
$$

For brevity, we omit the last common part $fmap\ local2global_M$ of these equations in the following proofs. Instead, we assume that the input is in the codomain of $fmap\ local2global_M$. Also, we use the condition in Theorem 6 that the input program does not use the *restore* operation.

For the first subcondition (3), we can define alg^{S}_{LHS} as follows.

$$
\begin{aligned}
& alg^{S}_{LHS} :: (Functor\,f, Undo\,s\,r) \Rightarrow Modify_F\,s\,r\,(s \to Free\,f\,[a]) \to (s \to Free\,f\,[a]) \\
& alg^{S}_{LHS}\,(MGet\,k) \;\;\;\;\;\;\;\; = \lambda s \to k\,s\,s \\
& alg^{S}_{LHS}\,(MUpdate\,r\,k) = \lambda s \to k\,(s \oplus r) \\
& alg^{S}_{LHS}\,(MRestore\,r\,k) = \lambda s \to k\,(s \ominus r)
\end{aligned}
$$

We prove it by a case analysis on the shape of input $op :: Modify_F\,s\,r\,(Free\,(Modify_F\,s\,r :+: Nondet_F :+: f)\,a)$. Note that we only need to consider the case that op is of form $MGet\,k$ or $MUpdate\,r\,k$ where *restore* is also not used in the continuation k.

case $op = MGet\,k$

$$
\begin{aligned}
& h_{GlobalM}\,(alg\,(Inl\,(MGet\,k))) \\
= \;& \{\text{- definition of } alg \text{ -}\} \\
& h_{GlobalM}\,(Op\,(Inl\,(MGet\,k))) \\
= \;& \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(Op\,(Inl\,(MGet\,k))))))\\
= \;& \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,(Op\,(Inr\,(Inl\,(fmap\,(\Leftrightarrow)\,(MGet\,k))))))) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,(Op\,(Inr\,(Inl\,(MGet\,((\Leftrightarrow) \circ k))))))) \\
= \;& \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(Op\,(fmap\,h_{ND+f}\,(Inl\,(MGet\,((\Leftrightarrow) \circ k))))))\\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(Op\,(Inl\,(MGet\,(h_{ND+f} \circ (\Leftrightarrow) \circ k)))))\\
= \;& \{\text{- definition of } h_{Modify1} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(\lambda s \to (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ k)\,s\,s) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to fmap\,fst\,((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ k)\,s\,s) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to ((fmap\,(fmap\,fst) \circ h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ k)\,s\,s) \\
= \;& \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& \lambda s \to (h_{GlobalM} \circ k)\,s\,s \\
= \;& \{\text{- definition of } alg^{S}_{LHS} \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& alg^{S}_{LHS}\,(MGet\,(h_{GlobalM} \circ k)) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& alg^{S}_{LHS}\,(fmap\,h_{GlobalM}\,(MGet\,k))
\end{aligned}
$$

case $op = MUpdate\ r\ k$ From op is in the codomain of $fmap\ local2global_M$ we obtain k is in the codomain of $local2global_M$.

$$
\begin{aligned}
& h_{GlobalM}\,(alg\,(Inl\,(MUpdate\,r\,k))) \\
= \;& \{\text{- definition of } alg \text{ -}\} \\
& h_{GlobalM}\,((update\,r \,[\!]\, side\,(restore\,r)) \gg k) \\
= \;& \{\text{- definitions of } side, update, restore, ([\!]), \text{and } (\gg) \text{ -}\} \\
& h_{GlobalM}\,(Op\,(Inr\,(Inl\,(Or\,(Op\,(Inl\,(MUpdate\,r\,k))) \\
& \qquad\qquad (Op\,(Inl\,(MRestore\,r\,(Op\,(Inr\,(Inl\,Fail))))))))))) \\
= \;& \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow) \\
& \quad (Op\,(Inr\,(Inl\,(Or\,(Op\,(Inl\,(MUpdate\,r\,k))) \\
& \qquad\qquad (Op\,(Inl\,(MRestore\,r\,(Op\,Inr\,((Inl\,Fail)))))))))))))) \\
= \;& \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,(\\
& \quad (Op\,(Inl\,(Or\,(Op\,(Inr\,(Inl\,(MUpdate\,r\,((\Leftrightarrow)\,k))))) \\
& \qquad\qquad (Op\,(Inr\,(Inl\,(MRestore\,r\,(Op\,(Inl\,Fail)))))))))))) \\
= \;& \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(\\
& \quad (liftM2\,(++)\,(Op\,(Inl\,(MUpdate\,r\,(h_{ND+f}\,((\Leftrightarrow)\,k))))) \\
& \qquad\qquad (Op\,(Inl\,(MRestore\,r\,(Var\,[\,]))))))) \\
= \;& \{\text{- definition of } liftM2 \text{ -}\} \\
& fmap\,(fmap\,fst)\,(h_{Modify1}\,(\\
& \quad \mathbf{do}\,x \leftarrow Op\,(Inl\,(MUpdate\,r\,(h_{ND+f}\,((\Leftrightarrow)\,k)))) \\
& \qquad y \leftarrow Op\,(Inl\,(MRestore\,r\,(Var\,[\,]))) \\
& \qquad Var\,(x ++ y) \\
&)) \\
= \;& \{\text{- Lemma 17 -}\} \\
& fmap\,(fmap\,fst)\,(\lambda t \rightarrow \\
& \quad \mathbf{do}\,(x, t_1) \leftarrow h_{Modify1}\,(Op\,(Inl\,(MUpdate\,r\,(h_{ND+f}\,((\Leftrightarrow)\,k)))))\,t \\
& \qquad (y, t_2) \leftarrow h_{Modify1}\,(Op\,(Inl\,(MRestore\,r\,(Var\,[\,]))))\,t_1 \\
& \qquad h_{Modify1}\,(Var\,(x ++ y))\,t_2 \\
&) \\
= \;& \{\text{- definition of } h_{Modify1} \text{ -}\} \\
& fmap\,(fmap\,fst) \\
& \quad (\lambda t \rightarrow \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))\,(t \oplus r) \\
& \qquad\qquad (y, t_2) \leftarrow Var\,([\,], t \ominus r) \\
& \qquad\qquad Var\,(x ++ y, t_2) \\
& \quad) \\
= \;& \{\text{- monad law -}\} \\
& fmap\,(fmap\,fst) \\
& \quad (\lambda t \rightarrow \mathbf{do}\,(x, t_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))\,(t \oplus r)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & \quad\quad\quad\quad Var\,(x +\!\!+ [\,], t_1 \ominus r) \\
 & \quad\quad) \\
= & \{\text{- right unit of } (+\!\!+) \text{ -}\} \\
 & fmap\,(fmap\,fst) \\
 & \quad\quad (\lambda t \to \mathbf{do}\,(x, t_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))\,(t \oplus r) \\
 & \quad\quad\quad\quad Var\,(x, t_1 \ominus r) \\
 & \quad\quad) \\
= & \{\text{- Lemma 16 -}\} \\
 & fmap\,(fmap\,fst) \\
 & \quad\quad (\lambda t \to \mathbf{do}\,(x, t \oplus r) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))\,(t \oplus r) \\
 & \quad\quad\quad\quad Var\,(x, (t \oplus r) \ominus r) \\
 & \quad\quad) \\
= & \{\text{- Equation (7.1) -}\} \\
 & fmap\,(fmap\,fst) \\
 & \quad\quad (\lambda t \to \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))\,(t \oplus r) \\
 & \quad\quad\quad\quad Var\,(x, t) \\
 & \quad\quad) \\
= & \{\text{- definition of } fmap\ fst \text{ -}\} \\
 & fmap\,(fmap\,fst) \\
 & \quad\quad (\lambda t \to \mathbf{do}\,x \leftarrow fmap\,fst\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))\,(t \oplus r)) \\
 & \quad\quad\quad\quad Var\,(x, t) \\
 & \quad\quad) \\
= & \{\text{- definition of } fmap \text{ -}\} \\
 & fmap\,(fmap\,fst) \\
 & \quad\quad (\lambda t \to \mathbf{do}\,x \leftarrow (fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))))\,(t \oplus r) \\
 & \quad\quad\quad\quad Var\,(x, t) \\
 & \quad\quad) \\
= & \{\text{- definition of } fmap\,(fmap\,fst) \text{ -}\} \\
 & \lambda t \to \mathbf{do}\,x \leftarrow (fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))))\,(t \oplus r) \\
 & \quad\quad Var\,x \\
= & \{\text{- monad law -}\} \\
 & \lambda t \to (fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,k))))\,(t \oplus r) \\
= & \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
 & \lambda t \to (h_{GlobalM}\,k)\,(t \oplus r) \\
= & \{\text{- definition of } alg^{S}_{LHS} \text{ -}\} \\
 & alg^{S}_{LHS}\,(MUpdate\,r\,(h_{GlobalM}\,k)) \\
= & \{\text{- definition of } fmap \text{ -}\} \\
 & alg^{S}_{LHS}\,(fmap\,h_{GlobalM}\,(MUpdate\,r\,k))
\end{array}
$$

For the second subcondition (4), we can define alg^{ND}_{LHS} as follows.

$$
\begin{array}{l}
alg^{ND}_{LHS} :: Functor\,f \Rightarrow Nondet_F\,(s \to Free\,f\,[a]) \to (s \to Free\,f\,[a]) \\
alg^{ND}_{LHS}\,Fail \quad\quad = \lambda s \to Var\,[\,] \\
alg^{ND}_{LHS}\,(Or\,p\,q) = \lambda s \to liftM2\,(+\!\!+)\,(p\,s)\,(q\,s)
\end{array}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

We prove it by a case analysis on the shape of input $op :: Nondet_F\ (Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)\ a)$.

case $op = Fail$

$$
\begin{aligned}
& h_{GlobalM}\ (alg\ (Inr\ (Inl\ Fail))) \\
= \ & \{\text{- definition of } alg \text{ -}\} \\
& h_{GlobalM}\ (Op\ (Inr\ (Inl\ Fail))) \\
= \ & \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inl\ Fail))))))) \\
= \ & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (h_{ND+f}\ (Op\ (Inl\ (fmap\ (\Leftrightarrow)\ Fail))))) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (h_{ND+f}\ (Op\ (Inl\ Fail)))) \\
= \ & \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (Var\ [\,])) \\
= \ & \{\text{- definition of } h_{Modify1} \text{ -}\} \\
& fmap\ (fmap\ fst)\ (\lambda s \to Var\ ([\,], s)) \\
= \ & \{\text{- definition of } fmap \text{ twice and } fst \text{ -}\} \\
& \lambda s \to Var\ [\,] \\
= \ & \{\text{- definition of } alg^{ND}_{RHS} \text{ -}\} \\
& alg^{ND}_{RHS}\ Fail \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& alg^{ND}_{RHS}\ (fmap\ h_{GlobalM}\ Fail)
\end{aligned}
$$

case $op = Or\ p\ q$ From op is in the codomain of $fmap\ local2global_M$, we obtain p and q are in the codomain of $local2global_M$.

$$
\begin{aligned}
& h_{GlobalM}\ (alg\ (Inr\ (Inl\ (Or\ p\ q)))) \\
= \ & \{\text{- definition of } alg \text{ -}\} \\
& h_{GlobalM}\ (Op\ (Inr\ (Inl\ (Or\ p\ q)))) \\
= \ & \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inl\ (Or\ p\ q))))))) \\
= \ & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (h_{ND+f}\ (Op\ (Inl\ (fmap\ (\Leftrightarrow)\ (Or\ p\ q)))))) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (h_{ND+f}\ (Op\ (Inl\ (Or\ ((\Leftrightarrow)\ p)\ ((\Leftrightarrow)\ q)))))) \\
= \ & \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (liftM2\ (+\!\!+)\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ (h_{ND+f}\ ((\Leftrightarrow)\ q)))) \\
= \ & \{\text{- definition of } liftM2 \text{ -}\} \\
& fmap\ (fmap\ fst)\ (h_{Modify1}\ (\mathbf{do}\ x \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ p) \\
& \qquad\qquad\qquad\qquad\qquad\quad y \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ q) \\
& \qquad\qquad\qquad\qquad\qquad\quad \eta\ (x +\!\!+ y))) \\
= \ & \{\text{- Lemma 17 -}\} \\
& fmap\ (fmap\ fst)\ (\lambda s_0 \to (\mathbf{do}\ (x, s_1) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_0 \\
& \qquad\qquad\qquad\qquad\qquad\quad\ (y, s_2) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s_1 \\
& \qquad\qquad\qquad\qquad\qquad\quad\ h_{Modify1}\ (\eta\ (x +\!\!+ y))\ s_2))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \ \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad fmap\,(fmap\,fst)\,(\lambda s_0 \to (\mathbf{do}\,(x, s_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p))\,s_0 \\
&\qquad\qquad\qquad\qquad\qquad\quad (y, s_2) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q))\,s_1 \\
&\qquad\qquad\qquad\qquad\qquad\quad Var\,(x +\!\!+ y, s_2))) \\
&= \ \{\text{- Lemma 16 -}\} \\
&\quad fmap\,(fmap\,fst)\,(\lambda s_0 \to (\mathbf{do}\,(x, s_1) \leftarrow \mathbf{do}\,\{(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p))\,s_0; \eta\,(x, s_0)\} \\
&\qquad\qquad\qquad\qquad\qquad\quad (y, s_2) \leftarrow \mathbf{do}\,\{(y, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q))\,s_1; \eta\,(x, s_1)\} \\
&\qquad\qquad\qquad\qquad\qquad\quad Var\,(x +\!\!+ y, s_2))) \\
&= \ \{\text{- monad laws -}\} \\
&\quad fmap\,(fmap\,fst)\,(\lambda s_0 \to (\mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p))\,s_0 \\
&\qquad\qquad\qquad\qquad\qquad\quad (y, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q))\,s_0 \\
&\qquad\qquad\qquad\qquad\qquad\quad Var\,(x +\!\!+ y, s_0))) \\
&= \ \{\text{- definition of } fmap \text{ (twice) and } fst \text{ -}\} \\
&\quad \lambda s_0 \to (\mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p))\,s_0 \\
&\qquad\qquad\quad (y, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q))\,s_0 \\
&\qquad\qquad\quad Var\,(x +\!\!+ y)) \\
&= \ \{\text{- definition of } fmap, fst \text{ and monad laws -}\} \\
&\quad \lambda s_0 \to (\mathbf{do}\,x \leftarrow fmap\,fst\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p))\,s_0) \\
&\qquad\qquad\quad y \leftarrow fmap\,fst\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q))\,s_0) \\
&\qquad\qquad\quad Var\,(x +\!\!+ y)) \\
&= \ \{\text{- definition of } fmap \text{ -}\} \\
&\quad \lambda s_0 \to (\mathbf{do}\,x \leftarrow fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p)))\,s_0 \\
&\qquad\qquad\quad y \leftarrow fmap\,(fmap\,fst)\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q)))\,s_0 \\
&\qquad\qquad\quad Var\,(x +\!\!+ y)) \\
&= \ \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
&\quad \lambda s_0 \to (\mathbf{do}\,x \leftarrow h_{GlobalM}\,p\,s_0 \\
&\qquad\qquad\quad y \leftarrow h_{GlobalM}\,q\,s_0 \\
&\qquad\qquad\quad Var\,(x +\!\!+ y)) \\
&= \ \{\text{- definition of } liftM2 \text{ -}\} \\
&\quad \lambda s_0 \to liftM2\,(+\!\!+)\,(h_{GlobalM}\,p\,s_0)\,(h_{GlobalM}\,q\,s_0) \\
&= \ \{\text{- definition of } alg^{ND}_{LHS} \text{ -}\} \\
&\quad alg^{ND}_{LHS}\,(Or\,(h_{GlobalM}\,p)\,(h_{GlobalM}\,q)) \\
&= \ \{\text{- definition of } fmap \text{ -}\} \\
&\quad alg^{ND}_{LHS}\,(fmap\,hGobal\,(Or\,p\,q))
\end{aligned}
$$

For the last subcondition (5), we can define fwd_{LHS} as follows.

$$
\begin{aligned}
&fwd_{LHS} :: Functor\,f \Rightarrow f\,(s \to Free\,f\,[a]) \to (s \to Free\,f\,[a]) \\
&fwd_{LHS}\,op = \lambda s \to Op\,(fmap\,(\$s)\,op)
\end{aligned}
$$

We prove it by the following calculation for input $op :: f\,(Free\,(Modify_F\,s\,r :+: Nondet_F :+: f)\,a)$.

$$
\begin{aligned}
&\quad h_{GlobalM}\,(alg\,(Inr\,(Inr\,op))) \\
&= \ \{\text{- definition of } alg \text{ -}\} \\
&\quad h_{GlobalM}\,(Op\,(Inr\,(Inr\,op))) \\
&= \ \{\text{- definition of } h_{GlobalM} \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ ((\Leftrightarrow)\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ \mathit{op}))))))\\
= \quad & \{\text{- definition of } (\Leftrightarrow) \text{ -}\}\\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{fmap}\ (\Leftrightarrow)\ \mathit{op}))))))\\
= \quad & \{\text{- definition of } h_{\mathit{ND+f}} \text{ -}\}\\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (\mathit{Op}\ (\mathit{fmap}\ h_{\mathit{ND+f}}\ (\mathit{Inr}\ (\mathit{fmap}\ (\Leftrightarrow)\ \mathit{op})))))\\
= \quad & \{\text{- definition of } \mathit{fmap} \text{ -}\}\\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{fmap}\ h_{\mathit{ND+f}}\ (\mathit{fmap}\ (\Leftrightarrow)\ \mathit{op})))))\\
= \quad & \{\text{- } \mathit{fmap} \text{ fusion -}\}\\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (\mathit{Op}\ (\mathit{Inr}\ (\mathit{fmap}\ (h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op}))))\\
= \quad & \{\text{- definition of } h_{\mathit{Modify1}} \text{ -}\}\\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s \rightarrow \mathit{Op}\ (\mathit{fmap}\ (\$s)\ (\mathit{fmap}\ h_{\mathit{Modify1}}\ (\mathit{fmap}\ (h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op}))))\\
= \quad & \{\text{- } \mathit{fmap} \text{ fusion -}\}\\
& \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s \rightarrow \mathit{Op}\ (\mathit{fmap}\ (\$s)\ (\mathit{fmap}\ (h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op})))\\
= \quad & \{\text{- definition of } \mathit{fmap} \text{ -}\}\\
& \lambda s \rightarrow \mathit{fmap}\ \mathit{fst}\ (\mathit{Op}\ (\mathit{fmap}\ (\$s)\ (\mathit{fmap}\ (h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op})))\\
= \quad & \{\text{- definition of } \mathit{fmap} \text{ -}\}\\
& \lambda s \rightarrow \mathit{Op}\ (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\mathit{fmap}\ (\$s)\ (\mathit{fmap}\ (h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op})))\\
= \quad & \{\text{- } \mathit{fmap} \text{ fusion -}\}\\
& \lambda s \rightarrow \mathit{Op}\ (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst} \circ (\$s))\ (\mathit{fmap}\ (h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op})))\\
= \quad & \{\text{- Lemma 2 -}\}\\
& \lambda s \rightarrow \mathit{Op}\ (\mathit{fmap}\ ((\$s) \circ \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}))\ (\mathit{fmap}\ (h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op})))\\
= \quad & \{\text{- } \mathit{fmap} \text{ fission -}\}\\
& \lambda s \rightarrow \mathit{Op}\ ((\mathit{fmap}\ (\$s) \circ \mathit{fmap}\ (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})))\ (\mathit{fmap}\ (h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op}))\\
= \quad & \{\text{- } \mathit{fmap} \text{ fusion -}\}\\
& \lambda s \rightarrow \mathit{Op}\ (\mathit{fmap}\ (\$s)\ (\mathit{fmap}\ (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst}) \circ h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{op}))\\
= \quad & \{\text{- definition of } h_{\mathit{GlobalM}} \text{ -}\}\\
& \lambda s \rightarrow \mathit{Op}\ (\mathit{fmap}\ (\$s)\ (\mathit{fmap}\ h_{\mathit{GlobalM}}\ \mathit{op}))\\
= \quad & \{\text{- definition of } \mathit{fwd}_{\mathit{LHS}} \text{ -}\}\\
& \mathit{fwd}_{\mathit{LHS}}\ (\mathit{fmap}\ h_{\mathit{GlobalM}}\ \mathit{op})
\end{aligned}
$$

6.4 Equating the fused sides

We observe that the following equations hold trivially.

$$
\begin{aligned}
\mathit{gen}_{\mathit{LHS}} &= \mathit{gen}_{\mathit{RHS}}\\
\mathit{alg}^{S}_{\mathit{LHS}} &= \mathit{alg}^{S}_{\mathit{RHS}}\\
\mathit{alg}^{\mathit{ND}}_{\mathit{LHS}} &= \mathit{alg}^{\mathit{ND}}_{\mathit{RHS}}\\
\mathit{fwd}_{\mathit{LHS}} &= \mathit{fwd}_{\mathit{RHS}}
\end{aligned}
$$

Therefore, the main theorem (Theorem 6) holds.

6.5 Key lemma: State restoration

Similar to Appendix 2, we have a key lemma saying that $\mathit{local2global}_M$ restores the initial state after a computation.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Lemma 16 (State is Restored)**.**
For any program $p :: Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)\ a$ *that do not use the operation* $OP\ (Inl\ MRestore\ \_\ \_)$*, we have*

$$\begin{aligned} & h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ p)))\ s \\ = \quad & \mathbf{do}\ (x, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ p)))\ s;\ \eta\ (x, s) \end{aligned}$$

Proof The proof follows the same structure of Lemma 1. We proceed by induction on t.

case $t = Var\ y$

$$\begin{aligned} & h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (Var\ y))))\ s \\ = \quad & \{\text{- definition of } local2global_M \text{ -}\} \\ & h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Var\ y)))\ s \\ = \quad & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\ & h_{Modify1}\ (h_{ND+f}\ (Var\ y))\ s \\ = \quad & \{\text{- definition of } h_{ND+f} \text{ -}\} \\ & h_{Modify1}\ (Var\ [y])\ s \\ = \quad & \{\text{- definition of } h_{Modify1} \text{ -}\} \\ & Var\ ([y], s) \\ = \quad & \{\text{- monad law -}\} \\ & \mathbf{do}\ (x, \_) \leftarrow Var\ ([y], s);\ Var\ (x, s) \\ = \quad & \{\text{- definition of } local2global_M, h_{ND+f}, (\Leftrightarrow), h_{Modify1} \text{ and } \eta \text{ -}\} \\ & \mathbf{do}\ (x, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (Var\ y))))\ s;\ \eta\ (x, s) \end{aligned}$$

case $t = Op\ (Inl\ (MGet\ k))$

$$\begin{aligned} & h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (Op\ (Inl\ (MGet\ k))))))\ s \\ = \quad & \{\text{- definition of } local2global_M \text{ -}\} \\ & h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inl\ (MGet\ (local2global_M \circ k))))))\ s \\ = \quad & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\ & h_{Modify1}\ (h_{ND+f}\ (Op\ (Inr\ (Inl\ (MGet\ ((\Leftrightarrow) \circ local2global_M \circ k))))))\ s \\ = \quad & \{\text{- definition of } h_{ND+f} \text{ -}\} \\ & h_{Modify1}\ (Op\ (Inl\ (MGet\ (h_{ND+f} \circ (\Leftrightarrow) \circ local2global_M \circ k))))\ s \\ = \quad & \{\text{- definition of } h_{Modify1} \text{ -}\} \\ & (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global_M \circ k)\ s\ s \\ = \quad & \{\text{- definition of } (\circ) \text{ -}\} \\ & (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (k\ s)))))\ s \\ = \quad & \{\text{- induction hypothesis -}\} \\ & \mathbf{do}\ (x, \_) \leftarrow h_{Modify1}\ ((\Leftrightarrow)\ (h_{ND+f}\ (local2global_M\ (k\ s))))\ s;\ \eta\ (x, s) \\ = \quad & \{\text{- definition of } local2global_M, (\Leftrightarrow), h_{ND+f}, h_{Modify1} \text{ -}\} \\ & \mathbf{do}\ (x, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ (local2global_M\ (Op\ (Inl\ (MGet\ k)))))\ s;\ \eta\ (x, s) \end{aligned}$$

case $t = Op\ (Inr\ (Inl\ Fail))$

$$\begin{aligned} & h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (Op\ (Inr\ (Inl\ Fail))))))\ s \\ = \quad & \{\text{- definition of } local2global_M \text{ -}\} \\ & h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inl\ Fail)))))\ s \\ = \quad & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \end{aligned}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_{Modify1}\,(h_{ND+f}\,(Op\,(Inl\,Fail)))\,s \\
= \;& \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& h_{Modify1}\,(Var\,[\,])\,s \\
= \;& \{\text{- definition of } h_{Modify1} \text{ -}\} \\
& Var\,([\,], s) \\
= \;& \{\text{- monad law -}\} \\
& \mathbf{do}\,(x, \_) \leftarrow Var\,([\,], s);\, Var\,(x, s) \\
= \;& \{\text{- definition of } local2global_M, (\Leftrightarrow), h_{ND+f}, h_{Modify1} \text{ -}\} \\
& \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,(Op\,(Inr\,(Inl\,Fail))))))\,s;\, \eta\,(x, s)
\end{aligned}
$$

case $t = Op\,(Inl\,(MUpdate\,r\,k))$

$$
\begin{aligned}
& h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,(Op\,(Inl\,(Put\,t\,k))))))\,s \\
= \;& \{\text{- definition of } local2global_M \text{ -}\} \\
& h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,((update\,r\,[\!]\,side\,(restore\,r)) \gg local2global_M\,k)))\,s \\
= \;& \{\text{- definition of } ([\!]), update, restore, side \text{ and } (\gg\!=) \text{ -}\} \\
& h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(\\
& \quad Op\,(Inr\,(Inl\,(Or\,(Op\,(Inl\,(MUpdate\,r\,(local2global_M\,k))))) \\
& \qquad\qquad (Op\,(Inl\,(MRestore\,r\,(Op\,(Inr\,(Inl\,Fail)))))))))))\,s \\
= \;& \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& h_{Modify1}\,(h_{ND+f}\,(\\
& \quad Op\,(Inl\,(Or\,(Op\,(Inr\,(Inl\,(MUpdate\,r\,((\Leftrightarrow)\,(local2global_M\,k))))))) \\
& \qquad (Op\,(Inr\,(Inl\,(MRestore\,r\,(Op\,(Inl\,Fail)))))))))\,s \\
= \;& \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& h_{Modify1}\,(\\
& \quad liftM2\,(+\!+)\,(Op\,(Inl\,(MUpdate\,r\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k))))))) \\
& \qquad (Op\,(Inl\,(MRestore\,r\,(Var\,[\,])))))\,s \\
= \;& \{\text{- definition of } liftM2 \text{ -}\} \\
& h_{Modify1}\,(\mathbf{do} \quad x \leftarrow Op\,(Inl\,(MUpdate\,r\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k))))) \\
& \qquad\qquad y \leftarrow Op\,(Inl\,(MRestore\,r\,(Var\,[\,]))) \\
& \qquad\qquad Var\,(x +\!+ y) \\
& \quad)\,s \\
= \;& \{\text{- Lemma 17 -}\} \\
& \mathbf{do}\,(x, s_1) \leftarrow h_{Modify1}\,(Op\,(Inl\,(MUpdate\,r\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k))))))\,s \\
& \quad (y, s_2) \leftarrow h_{Modify1}\,(Op\,(Inl\,(MRestore\,r\,(Var\,[\,]))))\,s_1 \\
& \quad Var\,(x +\!+ y, s_2) \\
= \;& \{\text{- definition of } h_{Modify1} \text{ -}\} \\
& \mathbf{do}\,(x, s_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k)))\,(s \oplus r) \\
& \quad (y, s_2) \leftarrow Var\,([\,], s_1 \ominus r) \\
& \quad Var\,(x +\!+ y, s_2) \\
= \;& \{\text{- monad laws -}\} \\
& \mathbf{do}\,(x, s_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k)))\,(s \oplus r) \\
& \quad Var\,(x +\!+ [\,], s_1 \ominus r) \\
= \;& \{\text{- right unit of } (+\!+) \text{ -}\} \\
& \mathbf{do}\,(x, s_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k)))\,(s \oplus r) \\
& \quad Var\,(x, s_1 \ominus r)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- induction hypothesis -}\}\\
&\quad \mathbf{do}\,(x, s_1) \leftarrow \mathbf{do}\,\{(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k)))\,(s \oplus r);\ \eta\,(x, s \oplus r)\}\\
&\qquad Var\,(x, s_1 \ominus r)\\
&= \quad \{\text{- monad laws -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k)))\,(s \oplus r)\\
&\qquad Var\,(x, (s \oplus r) \ominus r)\\
&= \quad \{\text{- Equation (7.1) -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k)))\,(s \oplus r)\\
&\qquad Var\,(x, s)\\
&= \quad \{\text{- monad laws -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow \mathbf{do}\,\{(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,k)))\,(s \oplus r);\ \eta\,(x, s)\}\\
&\qquad Var\,(x, s)\\
&= \quad \{\text{- deriviation in reverse -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,(Op\,(Inl\,(MUpdate\,r\,k))))))\,s\\
&\qquad Var\,(x, s)
\end{aligned}
$$

case $t = Op\,(Inr\,(Inl\,(Or\,p\,q)))$

$$
\begin{aligned}
&\quad h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,(Op\,(Inr\,(Inl\,(Or\,p\,q)))))))\,s\\
&= \quad \{\text{- definition of } local2global_M \text{ -}\}\\
&\quad h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(Op\,(Inr\,(Inl\,(Or\,(local2global_M\,p)\,(local2global_M\,q)))))))\,s\\
&= \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\}\\
&\quad h_{Modify1}\,(h_{ND+f}\,(Op\,(Inl\,(Or\,((\Leftrightarrow)\,(local2global_M\,p))\,((\Leftrightarrow)\,(local2global_M\,q))))))\,s\\
&= \quad \{\text{- definition of } h_{ND+f} \text{ -}\}\\
&\quad h_{Modify1}\,(liftM2\,(+\!\!+)\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,p)))\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,q))))\,s\\
&= \quad \{\text{- definition of } liftM2 \text{ -}\}\\
&\quad h_{Modify1}\,(\mathbf{do}\,x \leftarrow h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,p))\\
&\qquad\qquad\quad y \leftarrow h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,q))\\
&\qquad\qquad\quad Var\,(x +\!\!+ y)\\
&\qquad)\,s\\
&= \quad \{\text{- Lemma 17 -}\}\\
&\quad \mathbf{do}\,(x, s_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,p)))\,s\\
&\qquad (y, s_2) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,q)))\,s_1\\
&\qquad h_{Modify1}\,(Var\,(x +\!\!+ y))\,s_2\\
&= \quad \{\text{- induction hypothesis -}\}\\
&\quad \mathbf{do}\,(x, s_1) \leftarrow \mathbf{do}\,\{(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,p)))\,s;\ \eta\,(x, s)\}\\
&\qquad (y, s_2) \leftarrow \mathbf{do}\,\{(y, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,q)))\,s_1;\ \eta\,(y, s_1)\}\\
&\qquad h_{Modify1}\,(Var\,(x +\!\!+ y))\,s_2\\
&= \quad \{\text{- monad laws -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,p)))\,s\\
&\qquad (y, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,q)))\,s_1\\
&\qquad h_{Modify1}\,(Var\,(x +\!\!+ y))\,s\\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\}\\
&\quad \mathbf{do}\,(x, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,p)))\,s\\
&\qquad (y, \_) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2global_M\,q)))\,s\\
&\qquad \eta\,(x +\!\!+ y, s)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- monad laws -}\} \\
&\quad \mathbf{do}\ (x, \_) \leftarrow (\\
&\qquad \mathbf{do}\ (x, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ p)))\ s \\
&\qquad\quad (y, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ q)))\ s_1 \\
&\qquad\quad \eta\ (x +\!\!+ y, s) \\
&\qquad) \\
&\qquad \eta\ (x, s) \\
&= \quad \{\text{- derivation in reverse (similar to before) -}\} \\
&\quad \mathbf{do}\ (x, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (Op\ (Inr\ (Inl\ (Or\ p\ q)))))))\ s \\
&\qquad \eta\ (x, s)
\end{aligned}
$$

case $t = Op\ (Inr\ (Inr\ y))$

$$
\begin{aligned}
&\quad h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (Op\ (Inr\ (Inr\ y))))))\ s \\
&= \quad \{\text{- definition of } local2global_M \text{ -}\} \\
&\quad h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (Op\ (Inr\ (Inr\ (fmap\ local2global_M\ y))))))\ s \\
&= \quad \{\text{- definition of } (\Leftrightarrow);\ fmap \text{ fusion -}\} \\
&\quad h_{Modify1}\ (h_{ND+f}\ (Op\ (Inr\ (Inr\ (fmap\ ((\Leftrightarrow) \circ local2global_M)\ y)))))\ s \\
&= \quad \{\text{- definition of } h_{ND+f};\ fmap \text{ fusion -}\} \\
&\quad h_{Modify1}\ (Op\ (Inr\ (Inr\ (fmap\ (h_{ND+f} \circ (\Leftrightarrow) \circ local2global_M)\ y))))\ s \\
&= \quad \{\text{- definition of } h_{Modify1};\ fmap \text{ fusion -}\} \\
&\quad Op\ (Inr\ (Inr\ (fmap\ ((\$s) \circ h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global_M)\ y))) \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad Op\ (Inr\ (Inr\ (fmap\ ((\gg\!= \lambda(x, \_) \rightarrow \eta\ (x, s)) \circ (\$s) \circ h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global_M)\ y))) \\
&= \quad \{\text{- } fmap \text{ fission; definition of } (\gg\!=) \text{ -}\} \\
&\quad \mathbf{do}\ (x, \_) \leftarrow Op\ (Inr\ (Inr\ (fmap\ ((\$s) \circ h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2global_M)\ y))) \\
&\qquad \eta\ (x, s) \\
&= \quad \{\text{- deriviation in reverse (similar to before) -}\} \\
&\quad \mathbf{do}\ (x, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (local2global_M\ (Op\ (Inr\ (Inr\ y))))))\ s \\
&\qquad \eta\ (x, s)
\end{aligned}
$$

■

6.6 Auxiliary lemmas

The derivations above made use of several auxliary lemmas. We prove them here.

Lemma 17 (Distributivity of $h_{Modify1}$)**.**

$$h_{Modify1}\ (p \gg\!= k)\ s \quad = \quad h_{Modify1}\ p\ s \gg\!= \lambda(x, s') \rightarrow h_{Modify1}\ (k\ x)\ s'$$

Proof The proof follows the same structure of Lemma 4. We proceed by induction on p.

case $p = Var\ x$

$$
\begin{aligned}
&\quad h_{Modify1}\ (Var\ x \gg\!= k)\ s \\
&= \quad \{\text{- monad law -}\} \\
&\quad h_{Modify1}\ (k\ x)\ s
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- monad law -}\} \\
&\quad \eta\,(x, s) \gg\!= \lambda(x, s') \to h_{Modify1}\,(k\,x)\,s' \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad h_{Modify1}\,(Var\,x)\,s \gg\!= \lambda(x, s') \to h_{Modify1}\,(k\,x)\,s'
\end{aligned}
$$

case $p = Op\,(Inl\,(MGet\,p))$

$$
\begin{aligned}
&\quad h_{Modify1}\,(Op\,(Inl\,(MGet\,p)) \gg\!= k)\,s \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
&\quad h_{Modify1}\,(Op\,(fmap\,(\gg\!= k)\,(Inl\,(MGet\,p))))\,s \\
&= \quad \{\text{- definition of } fmap \text{ for coproduct } (:+:) \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(fmap\,(\gg\!= k)\,(MGet\,p))))\,s \\
&= \quad \{\text{- definition of } fmap \text{ for } MGet \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(MGet\,(\lambda x \to p\,s \gg\!= k))))\,s \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad h_{Modify1}\,(p\,s \gg\!= k)\,s \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad h_{Modify1}\,(p\,s)\,s \gg\!= \lambda(x, s') \to h_{Modify1}\,(k\,x)\,s' \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(MGet\,p)))\,s \gg\!= \lambda(x, s') \to h_{Modify1}\,(k\,x)\,s'
\end{aligned}
$$

case $p = Op\,(Inl\,(MUpdate\,r\,p))$

$$
\begin{aligned}
&\quad h_{Modify1}\,(Op\,(Inl\,(MUpdate\,r\,p)) \gg\!= k)\,s \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
&\quad h_{Modify1}\,(Op\,(fmap\,(\gg\!= k)\,(Inl\,(MUpdate\,r\,p))))\,s \\
&= \quad \{\text{- definition of } fmap \text{ for coproduct } (:+:) \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(fmap\,(\gg\!= k)\,(MUpdate\,r\,p))))\,s \\
&= \quad \{\text{- definition of } fmap \text{ for } MUpdate \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(MUpdate\,r\,(p \gg\!= k))))\,s \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad h_{Modify1}\,(p \gg\!= k)\,(s \oplus r) \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad h_{Modify1}\,p\,(s \oplus r) \gg\!= \lambda(x, s') \to h_{Modify1}\,(k\,x)\,s' \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(MUpdate\,r\,p)))\,s \gg\!= \lambda(x, s') \to h_{Modify1}\,(k\,x)\,s'
\end{aligned}
$$

case $p = Op\,(Inl\,(MRestore\,r\,p))$

$$
\begin{aligned}
&\quad h_{Modify1}\,(Op\,(Inl\,(MRestore\,r\,p)) \gg\!= k)\,s \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
&\quad h_{Modify1}\,(Op\,(fmap\,(\gg\!= k)\,(Inl\,(MRestore\,r\,p))))\,s \\
&= \quad \{\text{- definition of } fmap \text{ for coproduct } (:+:) \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(fmap\,(\gg\!= k)\,(MRestore\,r\,p))))\,s \\
&= \quad \{\text{- definition of } fmap \text{ for } MRestore \text{ -}\} \\
&\quad h_{Modify1}\,(Op\,(Inl\,(MRestore\,r\,(p \gg\!= k))))\,s \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad h_{Modify1}\,(p \gg\!= k)\,(s \ominus r)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad h_{Modify1}\ p\ (s \ominus r) \gg\!= \lambda(x, s') \to h_{Modify1}\ (k\ x)\ s' \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad h_{Modify1}\ (Op\ (Inl\ (MRestore\ r\ p)))\ s \gg\!= \lambda(x, s') \to h_{Modify1}\ (k\ x)\ s'
\end{aligned}
$$

case $p = Op\ (Inr\ y)$

$$
\begin{aligned}
&\quad h_{Modify1}\ (Op\ (Inr\ y) \gg\!= k)\ s \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ for free monad -}\} \\
&\quad h_{Modify1}\ (Op\ (fmap\ (\gg\!= k)\ (Inr\ y)))\ s \\
&= \quad \{\text{- definition of } fmap \text{ for coproduct } (:+:) \text{ -}\} \\
&\quad h_{Modify1}\ (Op\ (Inr\ (fmap\ (\gg\!= k)\ y)))\ s \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad Op\ (fmap\ (\lambda x \to h_{Modify1}\ x\ s)\ (fmap\ (\gg\!= k)\ y)) \\
&= \quad \{\text{- } fmap \text{ fusion -}\} \\
&\quad Op\ (fmap\ ((\lambda x \to h_{Modify1}\ (x \gg\!= k)\ s))\ y) \\
&= \quad \{\text{- induction hypothesis -}\} \\
&\quad Op\ (fmap\ (\lambda x \to h_{Modify1}\ x\ s \gg\!= \lambda(x', s') \to h_{Modify1}\ (k\ x')\ s')\ y) \\
&= \quad \{\text{- } fmap \text{ fission -}\} \\
&\quad Op\ (fmap\ (\lambda x \to x \gg\!= \lambda(x', s') \to h_{Modify1}\ (k\ x')\ s')\ (fmap\ (\lambda x \to h_{Modify1}\ x\ s)\ y)) \\
&= \quad \{\text{- definition of } (\gg\!=) \text{ -}\} \\
&\quad Op\ ((fmap\ (\lambda x \to h_{Modify1}\ x\ s)\ y)) \gg\!= \lambda(x', s') \to h_{Modify1}\ (k\ x')\ s' \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ -}\} \\
&\quad Op\ (Inr\ y)\ s \gg\!= \lambda(x', s') \to h_{Modify1}\ (k\ x')\ s'
\end{aligned}
$$

■

7 Proofs for modelling local state with trail stack

In this section, we prove the following theorem in Section 8.

Theorem 7. *Given Functor f and Undo s u, the equation*

$$h_{GlobalT} = h_{LocalM}$$

holds for all programs $p :: Free\ (Modify_F\ s\ u :+: Nondet_F :+: f)\ a$ *that do not use the operation* $Op\ (Inl\ (MRestore\ \_\ \_))$.

The proof follows a similar structure to those in Appendix 2 and Appendix 6.

As in Appendix 6, we fuse $run_{StateT} \circ h_{Modify}$ into $h_{Modify1}$ and use it instead in the following proofs.

7.1 Main proof structure

The main proof structure of Theorem 7 is as follows.

Proof The left-hand side is expanded to

$$h_{GlobalT} = fmap\ (fmap\ fst \circ flip\ run_{StateT}\ (Stack\ [\,]) \circ h_{State}) \circ h_{GlobalM} \circ local2trail$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Both the left-hand side and the right-hand side of the equation consist of function compositions involving one or more folds. We apply fold fusion separately on both sides to contract each into a single fold:

$$\begin{aligned} h_{GlobalT} &= fold\ gen_{LHS}\ (alg^{S}_{LHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{LHS}) \\ h_{LocalM} &= fold\ gen_{RHS}\ (alg^{S}_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS}) \end{aligned}$$

Finally, we show that both folds are equal by showing that their corresponding parameters are equal:

$$\begin{aligned} gen_{LHS} &= gen_{RHS} \\ alg^{S}_{LHS} &= alg^{S}_{RHS} \\ alg^{ND}_{LHS} &= alg^{ND}_{RHS} \\ fwd_{LHS} &= fwd_{RHS} \end{aligned}$$

We elaborate each of these steps below. ■

7.2 Fusing the right-hand side

We have already fused h_{LocalM} in Appendix 6.2. We just show the result here for easy reference.

$h_{LocalM} = fold\ gen_{RHS}\ (alg^{S}_{RHS} \,\#\, alg^{ND}_{RHS} \,\#\, fwd_{RHS})$
where
$gen_{RHS} :: Functor\ f \Rightarrow a \to (s \to Free\ f\ [a])$
$gen_{RHS}\ x = \lambda s \to Var\ [x]$
$alg^{S}_{RHS} :: Undo\ s\ r \Rightarrow State_F\ s\ (s \to p) \to (s \to p)$
$alg^{S}_{RHS}\ (MGet\ k) = \lambda s \to k\ s\ s$
$alg^{S}_{RHS}\ (MUpdate\ r\ k) = \lambda s \to k\ (s \oplus r)$
$alg^{S}_{RHS}\ (MRestore\ r\ k) = \lambda s \to k\ (s \oplus r)$
$alg^{ND}_{RHS} :: Functor\ f \Rightarrow Nondet_F\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a])$
$alg^{ND}_{RHS}\ Fail = \lambda s \to Var\ [\,]$
$alg^{ND}_{RHS}\ (Or\ p\ q) = \lambda s \to liftM2\ (+\!\!+)\ (p\ s)\ (q\ s)$
$fwd_{RHS} :: Functor\ f \Rightarrow f\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a])$
$fwd_{RHS}\ op = \lambda s \to Op\ (fmap\ (\$s)\ op)$

7.3 Fusing the left-hand side

As in Appendix 2, we fuse $run_{StateT} \circ h_{State}$ into h_{State1}. For brevity, we define

$runStack = fmap\ fst \circ flip\ h_{State1}\ (Stack\ [\,])$

The left-hand side is simplified to

$fmap\ runStack \circ h_{GlobalM} \circ local2trail$

We calculate as follows:

$fmap\ runStack \circ h_{GlobalM} \circ local2trail$
$=$ {- definition of $local2trail$ -}

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&\mathit{fmap}\ \mathit{runStack} \circ h_{\mathit{GlobalM}} \circ \mathit{fold}\ \mathit{Var}\ (\mathit{alg}_1 \,\#\, \mathit{alg}_2 \,\#\, \mathit{fwd}) \\
&\quad \textbf{where} \\
&\qquad \mathit{alg}_1\ (\mathit{MUpdate}\ r\ k) = \mathit{pushStack}\ (\mathit{Left}\ r) \gg \mathit{update}\ r \gg k \\
&\qquad \mathit{alg}_1\ p \qquad\qquad\ = \mathit{Op} \circ \mathit{Inl}\ \$\ p \\
&\qquad \mathit{alg}_2\ (\mathit{Or}\ p\ q) \quad = (\mathit{pushStack}\ (\mathit{Right}\ ()) \gg p) \,[\!]\, (\mathit{untrail} \gg q) \\
&\qquad \mathit{alg}_2\ p \qquad\qquad\ = \mathit{Op} \circ \mathit{Inr} \circ \mathit{Inl}\ \$\ p \\
&\qquad \mathit{fwd}\ p \qquad\qquad\ = \mathit{Op} \circ \mathit{Inr} \circ \mathit{Inr} \circ \mathit{Inr}\ \$\ p \\
&\qquad \mathit{untrail} = \textbf{do}\ \mathit{top} \leftarrow \mathit{popStack}; \\
&\qquad\qquad\qquad\ \textbf{case}\ \mathit{top}\ \textbf{of} \\
&\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \eta\ () \\
&\qquad\qquad\qquad\quad \mathit{Just}\ (\mathit{Right}\ ()) \rightarrow \eta\ () \\
&\qquad\qquad\qquad\quad \mathit{Just}\ (\mathit{Left}\ r) \rightarrow \textbf{do}\ \mathit{restore}\ r; \mathit{untrail} \\
&= \ \{\text{- fold fusion-post' (Equation 3.3) -}\} \\
&\quad \mathit{fold}\ \mathit{gen}_{\mathit{LHS}}\ (\mathit{alg}^{S}_{\mathit{LHS}} \,\#\, \mathit{alg}^{\mathit{ND}}_{\mathit{LHS}} \,\#\, \mathit{fwd}_{\mathit{LHS}})
\end{aligned}
$$

This last step is valid provided that the fusion conditions are satisfied:

$$\mathit{fmap}\ \mathit{runStack} \circ h_{\mathit{GlobalM}} \circ \mathit{Var} \quad = \quad \mathit{gen}_{\mathit{LHS}} \qquad (1)$$

$$
\begin{aligned}
&\quad \mathit{fmap}\ \mathit{runStack} \circ h_{\mathit{GlobalM}} \circ (\mathit{alg}_1 \,\#\, \mathit{alg}_2 \,\#\, \mathit{fwd}) \circ \mathit{fmap}\ \mathit{local2trail} \\
&= \ (\mathit{alg}^{S}_{\mathit{LHS}} \,\#\, \mathit{alg}^{\mathit{ND}}_{\mathit{LHS}} \,\#\, \mathit{fwd}_{\mathit{LHS}}) \circ \mathit{fmap}\ (\mathit{fmap}\ \mathit{runStack} \circ h_{\mathit{GlobalM}}) \circ \mathit{fmap}\ \mathit{local2trail} \qquad (2)
\end{aligned}
$$

The first subcondition (1) is met by

$$
\begin{aligned}
&\mathit{gen}_{\mathit{LHS}} :: \mathit{Functor}\ f \Rightarrow a \rightarrow (s \rightarrow \mathit{Free}\ f\ [a]) \\
&\mathit{gen}_{\mathit{LHS}}\ x = \lambda s \rightarrow \mathit{Var}\ [x]
\end{aligned}
$$

as established in the following calculation:

$$
\begin{aligned}
&\quad \mathit{fmap}\ \mathit{runStack}\ \$\ h_{\mathit{GlobalM}}\ (\mathit{Var}\ x) \\
&= \ \{\text{- definition of } h_{\mathit{GlobalM}} \text{ -}\} \\
&\quad \mathit{fmap}\ \mathit{runStack}\ \$\ \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND}+f}\ ((\Leftrightarrow)\ (\mathit{Var}\ x)))) \\
&= \ \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
&\quad \mathit{fmap}\ \mathit{runStack}\ \$\ \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND}+f}\ (\mathit{Var}\ x))) \\
&= \ \{\text{- definition of } h_{\mathit{ND}+f} \text{ -}\} \\
&\quad \mathit{fmap}\ \mathit{runStack}\ \$\ \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (h_{\mathit{Modify1}}\ (\mathit{Var}\ [x])) \\
&= \ \{\text{- definition of } h_{\mathit{Modify1}} \text{ -}\} \\
&\quad \mathit{fmap}\ \mathit{runStack}\ \$\ \mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ (\lambda s \rightarrow \mathit{Var}\ ([x], s)) \\
&= \ \{\text{- definition of } \mathit{fmap} \text{ (twice) -}\} \\
&\quad \mathit{fmap}\ \mathit{runStack}\ \$\ \lambda s \rightarrow \mathit{Var}\ [x] \\
&= \ \{\text{- definition of } \mathit{fmap} \text{ -}\} \\
&\quad \lambda s \rightarrow \mathit{runStack}\ \$\ \mathit{Var}\ [x] \\
&= \ \{\text{- definition of } \mathit{runStack} \text{ -}\} \\
&\quad \lambda s \rightarrow \mathit{fmap}\ \mathit{fst} \circ \mathit{flip}\ h_{\mathit{State1}}\ (\mathit{Stack}\ [\,])\ \$\ \mathit{Var}\ [x] \\
&= \ \{\text{- definition of } h_{\mathit{State1}} \text{ -}\} \\
&\quad \lambda s \rightarrow \mathit{fmap}\ \mathit{fst}\ \$\ (\lambda s \rightarrow \mathit{Var}\ ([x], s))\ (\mathit{Stack}\ [\,]) \\
&= \ \{\text{- function application -}\} \\
&\quad \lambda s \rightarrow \mathit{fmap}\ \mathit{fst}\ (\mathit{Var}\ ([x], \mathit{Stack}\ [\,]))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&= \{\text{- definition of } fmap \text{ -}\} \\
&\quad \lambda s \to Var\,[x] \\
&= \{\text{- definition of } gen_{LHS} \text{ -}\} \\
&\quad gen_{LHS}\, x
\end{aligned}
$$

We can split the second fusion condition (2) in three subconditions:

$$
\begin{aligned}
& fmap\ runStack \circ h_{GlobalM} \circ alg_1 \circ fmap\ local2trail \\
= \;& alg^{S}_{LHS} \circ fmap\ (fmap\ runStack \circ h_{GlobalM}) \circ fmap\ local2trail \qquad (3) \\
& fmap\ runStack \circ h_{GlobalM} \circ h_{GlobalM} \circ alg_2 \circ fmap\ local2trail \\
= \;& alg^{ND}_{LHS} \circ fmap\ (fmap\ runStack \circ h_{GlobalM}) \circ fmap\ local2trail \qquad (4) \\
& fmap\ runStack \circ h_{GlobalM} \circ h_{GlobalM} \circ fwd \circ fmap\ local2trail \\
= \;& fwd_{LHS} \circ fmap\ (fmap\ runStack \circ h_{GlobalM}) \circ fmap\ local2trail \qquad (5)
\end{aligned}
$$

For brevity, we omit the last common part $fmap\ local2global_M$ of these equations. Instead, we assume that the input is in the codomain of $fmap\ local2global_M$.

For the first subcondition (3), we define alg^{S}_{LHS} as follows.

$$
\begin{aligned}
&alg^{S}_{LHS} :: (Functor\, f, Undo\, s\, r) \Rightarrow Modify_F\ s\ r\ (s \to Free\, f\ [a]) \to (s \to Free\, f\ [a]) \\
&alg^{S}_{LHS}\ (MGet\ k) \qquad = \lambda s \to k\ s\ s \\
&alg^{S}_{LHS}\ (MUpdate\ r\ k) = \lambda s \to k\ (s \oplus r) \\
&alg^{S}_{LHS}\ (MRestore\ r\ k) = \lambda s \to k\ (s \ominus r)
\end{aligned}
$$

We prove it by a case analysis on the shape of input $op :: Modify_F\ s\ r\ (Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)\ a)$. We use the condition in Theorem 6 that the input program does not use the *restore* operation. We only need to consider the case that op is of form $MGet\ k$ or $MUpdate\ r\ k$, where *restore* is also not used in the continuation k.

case $op = MGet\ k$ In the corresponding case of Appendix 6.3, we have calculated that $h_{GlobalM}\ (Op\ (Inl\ (MGet\ k))) = \lambda s \to (h_{GlobalM} \circ k)\ s\ s$ $(\star)$.

$$
\begin{aligned}
&\quad fmap\ runStack\ \$\ h_{GlobalM}\ (alg_1\ (MGet\ k)) \\
&= \{\text{- definition of } alg_1 \text{ -}\} \\
&\quad fmap\ runStack\ \$\ h_{GlobalM}\ (Op\ (Inl\ (MGet\ k))) \\
&= \{\text{- Equation } (\star) \text{ -}\} \\
&\quad fmap\ runStack\ \$\ \lambda s \to (h_{GlobalM} \circ k)\ s\ s \\
&= \{\text{- definition of } fmap \text{ -}\} \\
&\quad \lambda s \to runStack\ \$\ (h_{GlobalM} \circ k)\ s\ s \\
&= \{\text{- definition of } fmap \text{ -}\} \\
&\quad \lambda s \to (fmap\ runStack \circ h_{GlobalM} \circ k)\ s\ s \\
&= \{\text{- definition of } alg^{S}_{LHS} \text{ -}\} \\
&\quad alg^{S}_{LHS}\ (MGet\ (fmap\ runStack \circ h_{GlobalM} \circ k)) \\
&= \{\text{- definition of } fmap \text{ -}\} \\
&\quad alg^{S}_{LHS}\ (fmap\ (fmap\ runStack \circ h_{GlobalM})\ (MGet\ k))
\end{aligned}
$$

case $op = MUpdate\ r\ k$ From op is in the codomain of $fmap\ local2global_M$ we obtain k is in the codomain of $local2global_M$.

$$
\begin{aligned}
&\quad fmap\ runStack \circ h_{GlobalM}\ \$\ alg_1\ (MUpdate\ r\ k) \\
&= \{\text{- definition of } alg_1 \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

```
  fmap runStack ∘ h_GlobalM $ pushStack (Left r) ≫ update r ≫ k
=   {- definition of pushStack -}
  fmap runStack ∘ h_GlobalM $ do
    Stack xs ← get
    put (Stack (Left r : xs))
    update r ≫ k
=   {- definition of get, put, and update -}
  fmap runStack ∘ h_GlobalM $
    Op ∘ Inr ∘ Inr ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inr ∘ Inr ∘ Inl $ Put (Stack (Left r : xs)) (
        Op ∘ Inl $ MUpdate r k))
=   {- definition of h_GlobalM -}
  fmap runStack ∘ fmap (fmap fst) ∘ h_Modify1 ∘ h_ND+f ∘ (⇔) $
    Op ∘ Inr ∘ Inr ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inr ∘ Inr ∘ Inl $ Put (Stack (Left r : xs)) (
        Op ∘ Inl $ MUpdate r k))
=   {- definition of (⇔) -}
  fmap runStack ∘ fmap (fmap fst) ∘ h_Modify1 ∘ h_ND+f $
    Op ∘ Inr ∘ Inr ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inr ∘ Inr ∘ Inl $ Put (Stack (Left r : xs)) (
        Op ∘ Inr ∘ Inl $ MUpdate r ((⇔) k)))
=   {- definition of h_ND+f -}
  fmap runStack ∘ fmap (fmap fst) ∘ h_Modify1 $
    Op ∘ Inr ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inr ∘ Inl $ Put (Stack (Left r : xs)) (
        Op ∘ Inl $ MUpdate r (h_ND+f ∘ (⇔) $ k)))
=   {- definition of h_Modify1 -}
  fmap runStack ∘ fmap (fmap fst) $ λs →
    Op ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inl $ Put (Stack (Left r : xs)) (
        (h_Modify1 ∘ h_ND+f ∘ (⇔) $ k) (s ⊕ r)))
=   {- definition of fmap (fmap fst) -}
  fmap runStack $ λs →
    Op ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inl $ Put (Stack (Left r : xs)) (
        (fmap (fmap fst) ∘ h_Modify1 ∘ h_ND+f ∘ (⇔) $ k) (s ⊕ r)))
=   {- definition of fmap -}
  λs → runStack $
    Op ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inl $ Put (Stack (Left r : xs)) (
        (fmap (fmap fst) ∘ h_Modify1 ∘ h_ND+f ∘ (⇔) $ k) (s ⊕ r)))
=   {- definition of h_GlobalM -}
  λs → runStack $
    Op ∘ Inl $ Get (λ(Stack xs) →
      Op ∘ Inl $ Put (Stack (Left r : xs)) (
```

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&\quad (h_{GlobalM}\ k)\ (s \oplus r)))\\
&= \ \{\text{- definition of } runStack \text{ -}\}\\
&\quad \lambda s \to fmap\ fst \circ flip\ h_{State1}\ (Stack\ [\,])\ \$\\
&\qquad Op \circ Inl\ \$\ Get\ (\lambda(Stack\ xs) \to\\
&\qquad\quad Op \circ Inl\ \$\ Put\ (Stack\ (Left\ r : xs))\ (\\
&\qquad\qquad (h_{GlobalM}\ k)\ (s \oplus r)))\\
&= \ \{\text{- definition of } h_{State1} \text{ -}\}\\
&\quad \lambda s \to fmap\ fst\ \$\ (\lambda t \to\\
&\qquad (\lambda(Stack\ xs) \to \lambda\_ \to\\
&\qquad\quad ((fmap\ h_{State1} \circ h_{GlobalM}\ \$\ k)\ (s \oplus r))\ (Stack\ (Left\ r : xs))\\
&\qquad)\ t\ t\\
&\quad)(Stack\ [\,])\\
&= \ \{\text{- function application -}\}\\
&\quad \lambda s \to fmap\ fst\ \$\\
&\qquad (\lambda(Stack\ xs) \to \lambda\_ \to\\
&\qquad\quad ((fmap\ h_{State1} \circ h_{GlobalM}\ \$\ k)\ (s \oplus r))\ (Stack\ (Left\ r : xs))\\
&\qquad)\ (Stack\ [\,])\ (Stack\ [\,])\\
&= \ \{\text{- function application -}\}\\
&\quad \lambda s \to fmap\ fst\ \$\\
&\qquad (\lambda\_ \to\\
&\qquad\quad ((fmap\ h_{State1} \circ h_{GlobalM}\ \$\ k)\ (s \oplus r))\ (Stack\ (Left\ r : [\,]))\\
&\qquad)\ (Stack\ [\,])\\
&= \ \{\text{- function application -}\}\\
&\quad \lambda s \to fmap\ fst\ \$\\
&\qquad ((fmap\ h_{State1} \circ h_{GlobalM}\ \$\ k)\ (s \oplus r))\ (Stack\ (Left\ r : [\,]))\\
&= \ \{\text{- function application -}\}\\
&\quad \lambda s \to fmap\ fst\ \$\\
&\qquad ((fmap\ h_{State1} \circ h_{GlobalM}\ \$\ k)\ (s \oplus r))\ (Stack\ (Left\ r : [\,]))\\
&= \ \{\text{- definition of } flip \text{ and reformulation -}\}\\
&\quad \lambda s \to (fmap\ (fmap\ fst \circ flip\ h_{State1}\ (Stack\ [Left\ r\,])) \circ h_{GlobalM}\ \$\ k)\ (s \oplus r)\\
&= \ \{\text{- Lemma 18 and definition of } fmap \text{ and } fst \text{ -}\}\\
&\quad \lambda s \to (fmap\ (fmap\ fst \circ flip\ h_{State1}\ (Stack\ [\,])) \circ h_{GlobalM}\ \$\ k)\ (s \oplus r)\\
&= \ \{\text{- definition of } runStack \text{ -}\}\\
&\quad \lambda s \to (fmap\ runStack \circ h_{GlobalM}\ \$\ k)\ (s \oplus r)\\
&= \ \{\text{- definition of } alg^{S}_{LHS} \text{ -}\}\\
&\quad alg^{S}_{LHS}\ (MUpdate\ r\ (fmap\ runStack \circ h_{GlobalM}\ \$\ k))\\
&= \ \{\text{- definition of } fmap \text{ -}\}\\
&\quad alg^{S}_{LHS}\ (fmap\ (fmap\ runStack \circ h_{GlobalM})\ (MUpdate\ r\ k))
\end{aligned}
$$

For the second subcondition (4), we can define alg^{ND}_{LHS} as follows.

$$
\begin{aligned}
&alg^{ND}_{LHS} :: Functor\ f \Rightarrow Nondet_F\ (s \to Free\ f\ [a]) \to (s \to Free\ f\ [a])\\
&alg^{ND}_{LHS}\ Fail \quad\ = \lambda s \to Var\ [\,]\\
&alg^{ND}_{LHS}\ (Or\ p\ q) = \lambda s \to liftM2\ (+\!\!+)\ (p\ s)\ (q\ s)
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

We prove it by a case analysis on the shape of input $op :: Nondet_F\ (Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)\ a)$.

case $op = Fail$ In the corresponding case of Appendix 6.3, we have calculated that $h_{GlobalM}\ (Op\ (Inr\ (Inl\ Fail))) = \lambda s \to Var\ [\,]$ $(\star)$.

$$
\begin{aligned}
& fmap\ runStack\ \$\ h_{GlobalM}\ (alg_2\ (Fail)) \\
= \quad & \{\text{- definition of } alg_2 \text{ -}\} \\
& fmap\ runStack\ \$\ h_{GlobalM}\ (Op\ (Inr\ (Inl\ Fail))) \\
= \quad & \{\text{- Equation } (\star) \text{ -}\} \\
& fmap\ runStack\ \$\ \lambda s \to Var\ [\,] \\
= \quad & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to runStack\ \$\ Var\ [\,] \\
= \quad & \{\text{- definition of } runStack \text{ -}\} \\
& \lambda s \to fmap\ fst \circ flip\ h_{State1}\ (Stack\ [\,])\ \$\ Var\ [\,] \\
= \quad & \{\text{- definition of } h_{State1} \text{ -}\} \\
& \lambda s \to fmap\ fst\ \$\ Var\ ([\,], Stack\ [\,]) \\
= \quad & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to Var\ [\,] \\
= \quad & \{\text{- definition of } alg^{ND}_{RHS} \text{ -}\} \\
& alg^{ND}_{RHS}\ Fail \\
= \quad & \{\text{- definition of } fmap \text{ -}\} \\
& alg^{ND}_{RHS}\ (fmap\ (fmap\ runStack \circ h_{GlobalM})\ Fail)
\end{aligned}
$$

case $op = Or\ p\ q$ From op is in the codomain of $fmap\ local2global_M$, we obtain p and q are in the codomain of $local2global_M$.

$$
\begin{aligned}
& fmap\ runStack \circ h_{GlobalM}\ \$\ alg_2\ (Or\ p\ q) \\
= \quad & \{\text{- definition of } alg_2 \text{ -}\} \\
& fmap\ runStack \circ h_{GlobalM}\ \$\ (pushStack\ (Right\ ()) \gg p)\ [\!]\ (untrail \gg q) \\
= \quad & \{\text{- definition of } ([\!]) \text{ -}\} \\
& fmap\ runStack \circ h_{GlobalM}\ \$\ Op \circ Inr \circ Inl\ \$\ Or \\
& \quad (pushStack\ (Right\ ()) \gg p)\ (untrail \gg q) \\
= \quad & \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& fmap\ runStack \circ fmap\ (fmap\ fst) \circ h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)\ \$\ Op \circ Inr \circ Inl\ \$\ Or \\
& \quad (pushStack\ (Right\ ()) \gg p)\ (untrail \gg q) \\
= \quad & \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& fmap\ runStack \circ fmap\ (fmap\ fst) \circ h_{Modify1} \circ h_{ND+f}\ \$\ Op \circ Inl\ \$\ Or \\
& \quad (pushStack\ (Right\ ()) \gg (\Leftrightarrow)\ p)\ (untrail \gg (\Leftrightarrow)\ q) \\
= \quad & \{\text{- definition of } h_{ND+f} \text{ and } liftM2 \text{ -}\} \\
& fmap\ runStack \circ fmap\ (fmap\ fst) \circ h_{Modify1}\ \$\ \mathbf{do} \\
& \quad x \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ()))) \gg h_{ND+f}\ ((\Leftrightarrow)\ p) \\
& \quad y \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ untrail) \gg h_{ND+f}\ ((\Leftrightarrow)\ q) \\
& \quad \eta\ (x +\!\!+ y) \\
= \quad & \{\text{- monad law -}\} \\
& fmap\ runStack \circ fmap\ (fmap\ fst) \circ h_{Modify1}\ \$\ \mathbf{do} \\
& \quad h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ())))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& x \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ p) \\
& h_{ND+f}\ ((\Leftrightarrow)\ untrail) \\
& y \leftarrow h_{ND+f}\ ((\Leftrightarrow)\ q) \\
& \eta\ (x +\!\!+ y) \\
= \ & \{\text{- definition of } h_{Modify1} \text{ and Lemma 17 -}\} \\
& fmap\ runStack \circ fmap\ (fmap\ fst)\ \$\ \lambda s \rightarrow \mathbf{do} \\
& (\_, s_1) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ()))))\ s \\
& (x, s_2) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_1 \\
& (\_, s3) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ untrail))\ s_2 \\
& (y, s4) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s3 \\
& \eta\ (x +\!\!+ y, s4) \\
= \ & \{\text{- definition of } fmap \text{ (twice) -}\} \\
& fmap\ runStack\ \$\ \lambda s \rightarrow \mathbf{do} \\
& (\_, s_1) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ()))))\ s \\
& (x, s_2) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_1 \\
& (\_, s3) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ untrail))\ s_2 \\
& (y, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s3 \\
& \eta\ (x +\!\!+ y) \\
= \ & \{\text{- definition of } fmap \text{ and } runStack \text{ -}\} \\
& \lambda s \rightarrow fmap\ fst \circ flip\ h_{State1}\ (Stack\ [\,])\ \$\ \mathbf{do} \\
& (\_, s_1) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ()))))\ s \\
& (x, s_2) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_1 \\
& (\_, s3) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ untrail))\ s_2 \\
& (y, \_) \leftarrow h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s3 \\
& \eta\ (x +\!\!+ y) \\
= \ & \{\text{- definition of } h_{State1} \text{ and Lemma 4 -}\} \\
& \lambda s \rightarrow fmap\ fst\ \$\ (\lambda t \rightarrow \mathbf{do} \\
& ((\_, s_1), t_1) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ()))))\ s)\ t \\
& ((x, s_2), t_2) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_1)\ t_1 \\
& ((\_, s3), t3) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ untrail))\ s_2)\ t_2 \\
& ((y, \_), t4) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s3)\ t3 \\
& \eta\ (x +\!\!+ y, t4))\ (Stack\ [\,]) \\
= \ & \{\text{- function application -}\} \\
& \lambda s \rightarrow fmap\ fst\ \$\ \mathbf{do} \\
& ((\_, s_1), t_1) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ()))))\ s)\ (Stack\ [\,]) \\
& ((x, s_2), t_2) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_1)\ t_1 \\
& ((\_, s3), t3) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ untrail))\ s_2)\ t_2 \\
& ((y, \_), t4) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s3)\ t3 \\
& \eta\ (x +\!\!+ y, t4) \\
= \ & \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \rightarrow \mathbf{do} \\
& ((\_, s_1), t_1) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ (pushStack\ (Right\ ()))))\ s)\ (Stack\ [\,]) \\
& ((x, s_2), t_2) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ p))\ s_1)\ t_1 \\
& ((\_, s3), t3) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ untrail))\ s_2)\ t_2 \\
& ((y, \_), \_) \leftarrow h_{State1}\ (h_{Modify1}\ (h_{ND+f}\ ((\Leftrightarrow)\ q))\ s3)\ t3
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \eta\,(x +\!\!+ y) \\
= \;& \{\text{- Lemma 20 -}\} \\
& \lambda s \to \mathbf{do} \\
& \quad ((x, \_), \_) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p))\,s)\,(Stack\,[\,]) \\
& \quad ((y, \_), \_) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q))\,s)\,(Stack\,[\,]) \\
& \quad \eta\,(x +\!\!+ y) \\
= \;& \{\text{- definition of } runStack \text{ -}\} \\
& \lambda s \to \mathbf{do} \\
& \quad (x, \_) \leftarrow runStack\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,p))\,s) \\
& \quad (y, \_) \leftarrow runStack\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,q))\,s) \\
& \quad \eta\,(x +\!\!+ y) \\
= \;& \{\text{- definition of } h_{GlobalM} \text{ -}\} \\
& \lambda s \to \mathbf{do} \\
& \quad x \leftarrow runStack\,(h_{GlobalM}\,p\,s) \\
& \quad y \leftarrow runStack\,(h_{GlobalM}\,q\,s) \\
& \quad \eta\,(x +\!\!+ y) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to \mathbf{do} \\
& \quad x \leftarrow (fmap\,runStack \circ hGobalM)\,p\,s \\
& \quad y \leftarrow (fmap\,runStack \circ hGobalM)\,q\,s \\
& \quad \eta\,(x +\!\!+ y)) \\
= \;& \{\text{- definition of } liftM2 \text{ -}\} \\
& \lambda s \to liftM2\,(+\!\!+)\,((fmap\,runStack \circ hGobalM)\,p\,s)\,((fmap\,runStack \circ hGobalM)\,q\,s) \\
= \;& \{\text{- definition of } alg^{ND}_{LHS} \text{ -}\} \\
& alg^{ND}_{LHS}\,(Or\,((fmap\,runStack \circ hGobalM)\,p)\,((fmap\,runStack \circ hGobalM)\,q)) \\
= \;& \{\text{- definition of } fmap \text{ -}\} \\
& alg^{ND}_{LHS}\,(fmap\,(fmap\,runStack \circ hGobalM)\,(Or\,p\,q))
\end{aligned}
$$

For the last subcondition (5), we can define fwd_{LHS} as follows.

$$
\begin{aligned}
& fwd_{LHS} :: Functor\,f \Rightarrow f\,(s \to Free\,f\,[a]) \to (s \to Free\,f\,[a]) \\
& fwd_{LHS}\,op = \lambda s \to Op\,(fmap\,(\$s)\,op)
\end{aligned}
$$

We prove it by the following calculation for input $op :: f\,(Free\,(Modify_F\,s\,r :+: Nondet_F :+: f)\,a)$. In the corresponding case of Appendix 6.3, we have calculated that $h_{GlobalM}\,(Op\,(Inr\,(Inr\,op))) = \lambda s \to Op\,(fmap\,(\$s)\,(fmap\,h_{GlobalM}\,op))$ $(\star)$.

$$
\begin{aligned}
& fmap\,runStack\,\$\,h_{GlobalM}\,(fwd\,op) \\
= \;& \{\text{- definition of } fwd \text{ -}\} \\
& fmap\,runStack\,\$\,h_{GlobalM}\,(Op \circ Inr \circ Inr \circ Inr\,\$\,op) \\
= \;& \{\text{- Equation } (\star) \text{ -}\} \\
& fmap\,runStack\,\$\,\lambda s \to Op\,(fmap\,(\$s)\,(fmap\,h_{GlobalM}\,(Inr\,op))) \\
= \;& \{\text{- fmap fusion -}\} \\
& fmap\,runStack\,\$\,\lambda s \to Op\,(fmap\,((\$s) \circ h_{GlobalM})\,(Inr\,op)) \\
= \;& \{\text{- reformulation -}\} \\
& fmap\,runStack\,\$\,\lambda s \to Op\,(fmap\,(\lambda x \to h_{GlobalM}\,x\,s)\,(Inr\,op)) \\
= \;& \{\text{- definition of } fmap \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& \lambda s \to runStack\ \$\ Op\ (fmap\ (\lambda x \to h_{GlobalM}\ x\ s)\ (Inr\ op)) \\
= & \quad \{\text{- definition of } runStack \text{ -}\} \\
& \lambda s \to fmap\ fst \circ flip\ h_{State1}\ (Stack\ [\,])\ \$ \\
& \quad Op\ (fmap\ (\lambda x \to h_{GlobalM}\ x\ s)\ (Inr\ op)) \\
= & \quad \{\text{- definition of } h_{State1} \text{ -}\} \\
& \lambda s \to fmap\ fst\ \$\ (\lambda t \to \\
& \quad Op\ (fmap\ (\$t) \circ fmap\ (h_{State1})\ \$\ fmap\ (\lambda x \to h_{GlobalM}\ x\ s)\ op))\ (Stack\ [\,]) \\
= & \quad \{\text{- fmap fusion and reformulation -}\} \\
& \lambda s \to fmap\ fst\ \$\ (\lambda t \to \\
& \quad Op\ (fmap\ (\lambda x \to h_{State1}\ (h_{GlobalM}\ x\ s)\ t)\ op))\ (Stack\ [\,]) \\
= & \quad \{\text{- function application -}\} \\
& \lambda s \to fmap\ fst\ \$ \\
& \quad Op\ (fmap\ (\lambda x \to h_{State1}\ (h_{GlobalM}\ x\ s)\ (Stack\ [\,]))\ op) \\
= & \quad \{\text{- definition of } fmap \text{ -}\} \\
& \lambda s \to Op\ (fmap\ (\lambda x \to fmap\ fst\ (h_{State1}\ (h_{GlobalM}\ x\ s)\ (Stack\ [\,])))\ op) \\
= & \quad \{\text{- reformulation -}\} \\
& \lambda s \to Op\ (fmap\ (\lambda x \to fmap\ (fmap\ fst \circ flip\ h_{State1}\ (Stack\ [\,])) \circ h_{GlobalM}\ \$\ x\ s)\ op) \\
= & \quad \{\text{- reformulation -}\} \\
& \lambda s \to Op\ (fmap\ (\lambda x \to (fmap\ runStack \circ h_{GlobalM}\ \$\ x)\ s)\ op) \\
= & \quad \{\text{- fmap fission -}\} \\
& \lambda s \to Op\ (fmap\ (\$s)\ (fmap\ (fmap\ runStack \circ h_{GlobalM})\ op)) \\
= & \quad \{\text{- definition of } fwd_{LHS} \text{ -}\} \\
& fwd_{LHS}\ (fmap\ (fmap\ runStack \circ h_{GlobalM})\ op)
\end{aligned}
$$

7.4 Equating the fused sides

We observe that the following equations hold trivially.

$$
\begin{aligned}
gen_{LHS} &= gen_{RHS} \\
alg^{S}_{LHS} &= alg^{S}_{RHS} \\
alg^{ND}_{LHS} &= alg^{ND}_{RHS} \\
fwd_{LHS} &= fwd_{RHS}
\end{aligned}
$$

Therefore, the main theorem (Theorem 7) holds.

7.5 Lemmas

In this section, we prove the lemmas used in Appendix 7.3.

The following lemma shows the relationship between the state and trail stack. Intuitively, the trail stack contains all the deltas (updates) that have not been restored in the program. Previous elements in the trail stack do not influence the result and state of programs.

Lemma 18 (Trail stack tracks state)**.** *For* $t :: Stack\ (Either\ r\ ())$, $s :: s$, *and* $p :: Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)$ *a which does not use the restore operation, we have*

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ t \\
= & \\
& \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ (Stack\ [\,]) \\
& \quad \eta\ ((x, fplus\ s\ ys), extend\ t\ ys)
\end{aligned}
$$

for some $ys = [Left\ r_n, ..., Left\ r\_1]$. *The functions extend and fplus are defined as follows:*

$$
\begin{aligned}
& extend :: Stack\ s \to [s] \to Stack\ s \\
& extend\ (Stack\ xs)\ ys = Stack\ (ys +\!\!+ xs) \\
& fplus :: Undo\ s\ r \Rightarrow s \to [Either\ r\ b] \to s \\
& fplus\ s\ ys = foldr\ (\lambda(Left\ r)\ s \to s \oplus r)\ s\ ys
\end{aligned}
$$

Note that an immediate corollary of Lemma 18 is that in addition to replacing the stack t with the empty stack $Stack\ [\,]$, we can also replace it with any other stack. The following equation holds.

$$
\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ t \\
= & \\
& \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ t' \\
& \quad \eta\ ((x, fplus\ s\ ys), extend\ t\ ys)
\end{aligned}
$$

We will also use this corollary in the proofs.

Proof

We proceed by induction on p.

case $p = Var\ y$

$$
\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (Var\ y))\ s)\ t \\
= & \quad \{\text{- definition of } local2trail \text{ -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (Var\ y)\ s)\ t \\
= & \quad \{\text{- definition of } (\Leftrightarrow) \text{ -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f})\ (Var\ y)\ s)\ t \\
= & \quad \{\text{- definition of } h_{ND+f} \text{ -}\} \\
& h_{State1}\ (h_{Modify1}\ (Var\ [y])\ s)\ t \\
= & \quad \{\text{- definition of } h_{Modify1} \text{ and functiona application -}\} \\
& h_{State1}\ (Var\ ([y], s))\ t \\
= & \quad \{\text{- definition of } h_{State1} \text{ and functiona application -}\} \\
& Var\ (([y], s), t) \\
= & \quad \{\text{- similar derivation in reverse -}\} \\
& \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (Var\ y))\ s)\ (Stack\ [\,]) \\
& \quad Var\ ((x, s), t)
\end{aligned}
$$

case $t = Op \circ Inr \circ Inl\ \$\ Fail$

$$
\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (Op \circ Inr \circ Inl\ \$\ Fail))\ s)\ t \\
= & \quad \{\text{- definition of } local2trail \text{ -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (Op \circ Inr \circ Inl\ \$\ Fail)\ s)\ t \\
= & \quad \{\text{- definition of } (\Leftrightarrow) \text{ and } h_{ND+f} \text{ -}\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
& h_{State1}\ (h_{Modify1}\ (Var\ [\,])\ s)\ t \\
= \ & \{\text{- definition of } h_{Modify1} \text{ and function application -}\} \\
& h_{State1}\ (Var\ ([\,], s))\ t \\
= \ & \{\text{- definition of } h_{State1} \text{ and functiona application -}\} \\
& Var\ (([\,], s), t) \\
= \ & \{\text{- similar derivation in reverse -}\} \\
& \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
& \qquad (local2trail\ (Op \circ Inr \circ Inl\ \$\ Fail))\ s)\ (Stack\ [\,]) \\
& \quad Var\ ((x, s), t)
\end{aligned}
$$

case $t = Op\ (Inl\ (MGet\ k))$

$$
\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (Op\ (Inl\ (MGet\ k))))\ s)\ t \\
= \ & \{\text{- definition of } local2trail \text{ -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (Op\ (Inl\ (MGet\ (local2trail \circ k))))\ s)\ t \\
= \ & \{\text{- definition of } (\Leftrightarrow) \text{ and } h_{ND+f} \text{ -}\} \\
& h_{State1}\ (h_{Modify1}\ (Op\ (Inl\ (MGet\ (h_{ND+f} \circ (\Leftrightarrow) \circ local2trail \circ k))))\ s)\ t \\
= \ & \{\text{- definition of } h_{Modify1} \text{ and function application -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail \circ k)\ s\ s)\ t \\
= \ & \{\text{- reformulation -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (k\ s))\ s)\ t \\
= \ & \{\text{- induction hypothesis on } k\ s \text{ -}\} \\
& \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (k\ s))\ s)\ (Stack\ [\,]) \\
& \quad \eta\ ((x, fplus\ s\ ys), extend\ t\ ys) \\
= \ & \{\text{- similar derivation in reverse -}\} \\
& \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
& \qquad (local2trail\ (Op\ (Inl\ (MGet\ k))))\ s)\ (Stack\ [\,]) \\
& \quad \eta\ ((x, fplus\ s\ ys), extend\ t\ ys)
\end{aligned}
$$

case $t = Op\ (Inl\ (MUpdate\ r\ k))$

$$
\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (Op\ (Inl\ (MUpdate\ r\ k))))\ s)\ t \\
= \ & \{\text{- definition of } local2trail \text{ -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (\mathbf{do} \\
& \quad pushStack\ (Left\ r) \\
& \quad update\ r \\
& \quad local2trail\ k \\
&)\ s)\ t \\
= \ & \{\text{- definition of } (\Leftrightarrow) \text{ and } h_{ND+f} \text{ -}\} \\
& h_{State1}\ (h_{Modify1}\ (\mathbf{do} \\
& \quad h_{ND+f} \circ (\Leftrightarrow)\ \$\ pushStack\ (Left\ r) \\
& \quad h_{ND+f} \circ (\Leftrightarrow)\ \$\ update\ r \\
& \quad h_{ND+f} \circ (\Leftrightarrow) \circ local2trail\ \$\ k \\
&)\ s)\ t \\
= \ & \{\text{- definition of } h_{Modify1}, \text{ Lemma 17 and function application -}\} \\
& h_{State1}\ (\mathbf{do} \\
& \quad (\_, s_1) \leftarrow (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)\ \$\ pushStack\ (Left\ r))\ s
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&\quad (\_, s_2) \leftarrow (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)\ \$\ update\ r)\ s_1 \\
&\quad (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail\ \$\ k)\ s_2 \\
&)\ t \\
&= \quad \{\text{- definition of } h_{Modify1} \text{ and } update \text{ -}\} \\
&\quad h_{State1}\ (\mathbf{do} \\
&\quad (\_, s_1) \leftarrow (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)\ \$\ pushStack\ (Left\ r))\ s \\
&\quad (\_, s_2) \leftarrow \eta\ ([\,()\,], s_1 \oplus r) \\
&\quad (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail\ \$\ k)\ s_2 \\
&)\ t \\
&= \quad \{\text{- monad law -}\} \\
&\quad h_{State1}\ (\mathbf{do} \\
&\quad (\_, s_1) \leftarrow (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)\ \$\ pushStack\ (Left\ r))\ s \\
&\quad (h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail\ \$\ k)\ (s_1 \oplus r) \\
&)\ t \\
&= \quad \{\text{- definition of } h_{State1}, \text{ Lemma 4, and function application -}\} \\
&\quad \mathbf{do}\ ((\_, s_1), t_1) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)\ \$\ pushStack\ (Left\ r))\ s)\ t \\
&\qquad h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail\ \$\ k)\ (s_1 \oplus r))\ t_1 \\
&= \quad \{\text{- definition of } pushStack \text{ -}\} \\
&\quad \mathbf{do}\ ((\_, s_1), t_1) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)\ \$\ \mathbf{do} \\
&\qquad\quad Stack\ xs \leftarrow get \\
&\qquad\quad put\ (Stack\ (Left\ r : xs)))\ s)\ t \\
&\qquad h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail\ \$\ k)\ (s_1 \oplus r))\ t_1 \\
&= \quad \{\text{- definition of } h_{State1}, h_{Modify1}, h_{ND+f}, (\Leftrightarrow), get, \text{ and } put\text{; let } t = Stack\ xs \text{ -}\} \\
&\quad \mathbf{do}\ ((\_, s_1), t_1) \leftarrow \eta\ (([\,()\,], s), Stack\ (Left\ r : xs)) \\
&\qquad h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ k)\ (s_1 \oplus r))\ t_1 \\
&= \quad \{\text{- monad law -}\} \\
&\quad h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ k)\ (s \oplus r))\ (Stack\ (Left\ r : xs)) \\
&= \quad \{\text{- by induction hypothesis on } k, \text{ for some } ys \text{ the equation holds -}\} \\
&\quad \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
&\qquad\quad (local2trail\ k)\ (s \oplus r))\ (Stack\ [Left\ r]) \\
&\qquad \eta\ ((x, fplus\ (s \oplus r)\ ys), extend\ (Stack\ (Left\ r : xs))\ ys) \\
&= \quad \{\text{- definition of } fplus \text{ and } extend \text{ -}\} \\
&\quad \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
&\qquad\quad (local2trail\ k)\ (s \oplus r))\ (Stack\ [Left\ r]) \\
&\qquad \eta\ ((x, fplus\ s\ (ys +\!\!+ [Left\ r])), extend\ (Stack\ xs)\ (ys +\!\!+ [Left\ r])) \\
&= \quad \{\text{- let } ys' = ys +\!\!+ [Left\ r]\text{; Equation } t = Stack\ xs \text{ -}\} \\
&\quad \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
&\qquad\quad (local2trail\ k)\ (s \oplus r))\ (Stack\ [Left\ r]) \\
&\qquad \eta\ ((x, fplus\ s\ ys'), extend\ t\ ys') \\
&= \quad \{\text{- similar derivation in reverse -}\} \\
&\quad \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
&\qquad\quad (local2trail\ (Op\ (Inl\ (MUpdate\ r\ k))))\ s)\ (Stack\ [\,]) \\
&\qquad \eta\ ((x, fplus\ s\ ys'), extend\ t\ ys')
\end{aligned}
$$

case $t = Op \circ Inr \circ Inl\ \$\ Or\ p\ q$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$h_{State1}\,((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\,(local2trail\,(Op \circ Inr \circ Inl\ \$\ Or\ p\ q))\,s)\,t$
$=$ {- definition of *local2trail* -}
$h_{State1}\,((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\,($
$(pushStack\,(Right\,()) \gg local2trail\,p)\ [\!]\ (untrail \gg local2trail\,q))\,s)\,t$
$=$ {- definition of $([\!])$ -}
$h_{State1}\,((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\,(Op \circ Inr \circ Inl\ \$\ Or$
$(pushStack\,(Right\,()) \gg local2trail\,p)$
$(untrail \gg local2trail\,q))\,s)\,t$
$=$ {- definition of h_{ND+f}, $(\Leftrightarrow)$, and *liftM2* -}
$h_{State1}\,(h_{Modify1}\,(\mathbf{do}$
$x \leftarrow h_{ND+f}\,((\Leftrightarrow)\,(pushStack\,(Right\,()))) \gg h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,p))$
$y \leftarrow h_{ND+f}\,((\Leftrightarrow)\,untrail) \gg h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,q))$
$\eta\,(x +\!\!+ y)$
$)s)\,t$
$=$ {- monad law -}
$h_{State1}\,(h_{Modify1}\,(\mathbf{do}$
$h_{ND+f}\,((\Leftrightarrow)\,(pushStack\,(Right\,())))$
$x \leftarrow h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,p))$
$h_{ND+f}\,((\Leftrightarrow)\,untrail)$
$y \leftarrow h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,q))$
$\eta\,(x +\!\!+ y)$
$)s)\,t$
$=$ {- definition of $h_{Modify1}$, Lemma 17, and function application -}
$h_{State1}\,(\mathbf{do}$
$(\_, s_1) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(pushStack\,(Right\,()))))\,s$
$(x, s_2) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,p)))\,s_1$
$(\_, s3) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,untrail))\,s_2$
$(y, s4) \leftarrow h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,q)))\,s3$
$\eta\,(x +\!\!+ y, s4)$
$)\,t$
$=$ {- definition of h_{State1}, Lemma 4, and function application -}
$\mathbf{do}\,((\_, s_1), t_1) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(pushStack\,(Right\,()))))\,s)\,t$
$((x, s_2), t_2) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,p)))\,s_1)\,t_1$
$((\_, s3), t3) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,untrail))\,s_2)\,t_2$
$((y, s4), t4) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,q)))\,s3)\,t3$
$\eta\,((x +\!\!+ y, s4), t4)$
$=$ {- definition of *pushStack* -}
$\mathbf{do}\,((\_, s_1), t_1) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(\mathbf{do}$
$Stack\,xs \leftarrow get$
$put\,(Stack\,(Right\,() : xs)))))\,s)\,t$
$((x, s_2), t_2) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,p)))\,s_1)\,t_1$
$((\_, s3), t3) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,untrail))\,s_2)\,t_2$
$((y, s4), t4) \leftarrow h_{State1}\,(h_{Modify1}\,(h_{ND+f}\,((\Leftrightarrow)\,(local2trail\,q)))\,s3)\,t3$
$\eta\,((x +\!\!+ y, s4), t4)$
$=$ {- definition of h_{State1}, $h_{Modify1}$, h_{ND+f}, $(\Leftrightarrow)$, *get*, *put*; let $t = Stack\,xs$ -}

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{aligned}
&\textbf{do}\ ((\_, s_1), t_1) \leftarrow \eta\ (([()], s), \mathit{Stack}\ (\mathit{Right}\ () : xs))\\
&\quad ((x, s_2), t_2) \leftarrow h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ ((\Leftrightarrow)\ (\mathit{local2trail}\ p)))\ s_1)\ t_1\\
&\quad ((\_, s3), t3) \leftarrow h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ ((\Leftrightarrow)\ \mathit{untrail}))\ s_2)\ t_2\\
&\quad ((y, s4), t4) \leftarrow h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ ((\Leftrightarrow)\ (\mathit{local2trail}\ q)))\ s3)\ t3\\
&\quad \eta\ ((x +\!\!+ y, s4), t4)\\
=\ &\{\text{- monad law -}\}\\
&\textbf{do}\ ((x, s_2), t_2) \leftarrow h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ ((\Leftrightarrow)\\
&\qquad\qquad (\mathit{local2trail}\ p)))\ s)\ (\mathit{Stack}\ (\mathit{Right}\ () : xs))\\
&\quad ((\_, s3), t3) \leftarrow h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ ((\Leftrightarrow)\ \mathit{untrail}))\ s_2)\ t_2\\
&\quad ((y, s4), t4) \leftarrow h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND+f}}\ ((\Leftrightarrow)\ (\mathit{local2trail}\ q)))\ s3)\ t3\\
&\quad \eta\ ((x +\!\!+ y, s4), t4)\\
=\ &\{\text{- reformulation -}\}\\
&\textbf{do}\ ((x, s_2), t_2) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ p)\ s)\ (\mathit{Stack}\ (\mathit{Right}\ () : xs))\\
&\quad ((\_, s3), t3) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ \mathit{untrail}\ s_2)\ t_2\\
&\quad ((y, s4), t4) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ (\mathit{local2trail}\ q)\ s3)\ t3\\
&\quad \eta\ ((x +\!\!+ y, s4), t4)\\
=\ &\{\text{- by induction hypothesis on } p\text{, for some } ys \text{ the equation holds -}\}\\
&\textbf{do}\ ((x, \_), \_) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ p)\ s)\ (\mathit{Stack}\ (\mathit{Right}\ ()))\\
&\quad ((\_, s3), t3) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad \mathit{untrail}\ (\mathit{fplus}\ s\ ys))\ (\mathit{extend}\ (\mathit{Stack}\ (\mathit{Right}\ () : xs))\ ys)\\
&\quad ((y, s4), t4) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\ (\mathit{local2trail}\ q)\ s3)\ t3\\
&\quad \eta\ ((x +\!\!+ y, s4), t4)\\
=\ &\{\text{- Lemma 19 -}\}\\
&\textbf{do}\ ((x, \_), \_) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ p)\ s)\ (\mathit{Stack}\ (\mathit{Right}\ ()))\\
&\quad ((y, s4), t4) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ q)\ (\mathit{fminus}\ (\mathit{fplus}\ s\ ys)\ ys))\ (\mathit{Stack}\ xs)\\
&\quad \eta\ ((x +\!\!+ y, s4), t4)\\
=\ &\{\text{- Equation (7.1) gives } \mathit{fminus}\ (\mathit{fplus}\ s\ ys)\ ys = s \text{ -}\}\\
&\textbf{do}\ ((x, \_), \_) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ p)\ s)\ (\mathit{Stack}\ (\mathit{Right}\ ()))\\
&\quad ((y, s4), t4) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ q)\ s)\ (\mathit{Stack}\ xs)\\
&\quad \eta\ ((x +\!\!+ y, s4), t4)\\
=\ &\{\text{- by induction hypothesis on } p\text{, for some } ys' \text{ the equation holds -}\}\\
&\textbf{do}\ ((x, \_), \_) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ p)\ s)\ (\mathit{Stack}\ (\mathit{Right}\ ()))\\
&\quad ((y, \_), \_) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))\\
&\qquad\qquad (\mathit{local2trail}\ q)\ s)\ (\mathit{Stack}\ [\,])\\
&\quad \eta\ ((x +\!\!+ y, \mathit{fplus}\ s\ ys'), \mathit{extend}\ (\mathit{Stack}\ xs)\ ys')\\
=\ &\{\text{- similar derivation in reverse -}\}\\
&\textbf{do}\ ((x, \_), \_) \leftarrow h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND+f}} \circ (\Leftrightarrow))
\end{aligned}
$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$(local2trail\ (Op \circ Inr \circ Inl\ \$\ Or\ p\ q))\ s)\ (Stack\ [\,])$$
$$\eta\ ((x, fplus\ s\ ys'), extend\ t\ ys')$$

case $t = Op \circ Inr \circ Inr\ \$\ y$

$$\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (Op \circ Inr \circ Inr\ \$\ y))\ s)\ t \\
= \quad & \{\text{- definition of } local2trail \text{ -}\} \\
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (Op \circ Inr \circ Inr \circ Inr \circ fmap\ local2trail\ \$\ y)\ s)\ t \\
= \quad & \{\text{- definition of } (\Leftrightarrow) \text{ and } h_{ND+f} \text{ -}\} \\
& h_{State1}\ (h_{Modify1}\ (Op \circ Inr \circ Inr \circ fmap\ (h_{ND+f} \circ (\Leftrightarrow) \circ local2trail)\ \$\ y)\ s)\ t \\
= \quad & \{\text{- definition of } h_{Modify1} \text{ and function application -}\} \\
& h_{State1}\ (Op \circ Inr \circ fmap\ (\$s) \circ h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail)\ \$\ y)\ t \\
= \quad & \{\text{- definition of } h_{State1} \text{ and function application -}\} \\
& Op \circ fmap\ (\$t) \circ h_{State1} \circ (\$s) \circ h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow) \circ local2trail)\ \$\ y \\
= \quad & \{\text{- reformulation -}\} \\
& Op \circ fmap\ (\lambda k \rightarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ k)\ s)\ t)\ \$\ y \\
= \quad & \{\text{- by induction hypothesis on } y, \text{ for some } ys \text{ the equation holds -}\} \\
& Op \circ fmap\ (\lambda k \rightarrow \mathbf{do} \\
& \quad ((x, _), _) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ k)\ s)\ t \\
& \quad \eta\ ((x, fplus\ s\ ys), extend\ t\ ys))\ \$\ y \\
= \quad & \{\text{- definition of free monad -}\} \\
& \mathbf{do}\ ((x, _), _) \leftarrow Op \circ fmap\ (\lambda k \rightarrow \\
& \quad h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ k)\ s)\ t)\ \$\ y \\
& \quad \eta\ ((x, fplus\ s\ ys), extend\ t\ ys) \\
= \quad & \{\text{- similar derivation in reverse -}\} \\
& \mathbf{do}\ ((x, _), _) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ (Op \circ Inr \circ Inr\ \$\ y))\ s)\ t \\
& \quad \eta\ ((x, fplus\ s\ ys), extend\ t\ ys)
\end{aligned}$$

■

The following lemma shows that the *untrail* function restores all the updates in the trail stack until it reaches a time stamp *Right* ().

Lemma 19 (UndoTrail undos). *For* $t = Stack\ (ys +\!\!+ (Right\ () : xs))$ *and* $ys = [Left\ r_1, ..., Left\ r_n]$, *we have*

$$\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ untrail\ s)\ t \\
= \quad & \\
& \eta\ (([\,()\,], fminus\ s\ ys), Stack\ xs)
\end{aligned}$$

The function fminus is defined as follows:

$$\begin{aligned}
& fminus :: Undo\ s\ r \Rightarrow s \rightarrow [Either\ r\ b] \rightarrow s \\
& fminus\ s\ ys = foldl\ (\lambda s\ (Left\ r) \rightarrow s \ominus r)\ s\ ys
\end{aligned}$$

Proof

We first calculate as follows:

$$\begin{aligned}
& h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ untrail\ s)\ t \\
= \quad & \{\text{- definition of } untrail \text{ -}\}
\end{aligned}$$

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

```
  hState1 ((hModify1 ∘ hND+f ∘ (⇔)) (do
    top ← popStack
    case top of
      Nothing → η ()
      Just (Right ()) → η ()
      Just (Left r) → do restore r; untrail
  )s) t
=  {- definition of popStack -}
  hState1 ((hModify1 ∘ hND+f ∘ (⇔)) (do
    top ← do Stack xs ← get
             case xs of
               [ ]       → η Nothing
               (x : xs′) → do put (Stack xs′); η (Just x)
    case top of
      Nothing → η ()
      Just (Right ()) → η ()
      Just (Left r) → do restore r; untrail
  )s) t
=  {- monad law and case split -}
  hState1 ((hModify1 ∘ hND+f ∘ (⇔)) (do
    Stack xs ← get
    case xs of
      [ ]             → η ()
      (Right () : xs′) → do put (Stack xs′); η ()
      (Left r : xs′)   → do put (Stack xs′); restore r; untrail
  )s) t
=  {- definition of get -}
  hState1 ((hModify1 ∘ hND+f ∘ (⇔)) (Op ∘ Inr ∘ Inr ∘ Inl ∘ Get $ λ(Stack xs) →
    case xs of
      [ ]             → η ()
      (Right () : xs′) → do put (Stack xs′); η ()
      (Left r : xs′)   → do put (Stack xs′); restore r; untrail
  )s) t
=  {- definition of hND+f and (⇔) -}
  hState1 (hModify1 (Op ∘ Inr ∘ Inl ∘ Get $ λ(Stack xs) →
    case xs of
      [ ]             → η [()]
      (Right () : xs′) → hND+f ∘ (⇔) $ do put (Stack xs′); η ()
      (Left r : xs′)   → hND+f ∘ (⇔) $ do put (Stack xs′); restore r; untrail
  )s) t
=  {- definition of hModify1 and function application -}
  hState1 (Op ∘ Inl ∘ Get $ λ(Stack xs) →
    case xs of
      [ ]             → η ([()], s)
      (Right () : xs′) → hModify1 (hND+f ∘ (⇔) $ do put (Stack xs′); η ()) s
```

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

$$
\begin{array}{l}
\quad\quad (\mathit{Left}\ r : xs') \quad \to h_{\mathit{Modify1}}\ (h_{\mathit{ND}+f} \circ (\Leftrightarrow)\ \$\ \mathbf{do}\ \mathit{put}\ (\mathit{Stack}\ xs'); \mathit{restore}\ r; \mathit{untrail})\ s \\
)\ t \\
= \quad \{\text{- definition of } h_{\mathit{State1}} \text{ and function application; let } t = \mathit{Stack}\ (ys +\!\!+ (\mathit{Right}\ () : xs)) \text{ -}\} \\
\quad\quad \mathbf{case}\ (ys +\!\!+ (\mathit{Right}\ () : xs))\ \mathbf{of} \\
\quad\quad\quad [\,] \quad\quad\quad\quad \to \eta\ (([\,()\,], s), t) \\
\quad\quad\quad (\mathit{Right}\ () : xs') \to \eta\ (([\,()\,], s), \mathit{Stack}\ xs') \\
\quad\quad\quad (\mathit{Left}\ r : xs') \quad \to h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND}+f} \circ (\Leftrightarrow)\ \$ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{do}\ \mathit{put}\ (\mathit{Stack}\ xs'); \mathit{restore}\ r; \mathit{untrail})\ s)\ t
\end{array}
$$

Then, we proceed by an induction on the structure of ys.

case $ys = [\,]$

$$
\begin{array}{l}
\quad\quad \mathbf{case}\ (ys +\!\!+ (\mathit{Right}\ () : xs))\ \mathbf{of} \\
\quad\quad\quad [\,] \quad\quad\quad\quad \to \eta\ (([\,()\,], s), t) \\
\quad\quad\quad (\mathit{Right}\ () : xs') \to \eta\ (([\,()\,], s), \mathit{Stack}\ xs') \\
\quad\quad\quad (\mathit{Left}\ r : xs') \quad \to h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND}+f} \circ (\Leftrightarrow)\ \$ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{do}\ \mathit{put}\ (\mathit{Stack}\ xs'); \mathit{restore}\ r; \mathit{untrail})\ s)\ t \\
= \quad \{\text{- case split -}\} \\
\quad \eta\ (([\,()\,], s), \mathit{Stack}\ xs) \\
= \quad \{\text{- definition of } \mathit{fminus} \text{ -}\} \\
\quad \eta\ (([\,()\,], \mathit{fminus}\ s\ [\,]), \mathit{Stack}\ xs)
\end{array}
$$

case $ys = (\mathit{Left}\ r : ys')$

$$
\begin{array}{l}
\quad\quad \mathbf{case}\ (ys +\!\!+ (\mathit{Right}\ () : xs))\ \mathbf{of} \\
\quad\quad\quad [\,] \quad\quad\quad\quad \to \eta\ (([\,()\,], s), t) \\
\quad\quad\quad (\mathit{Right}\ () : xs') \to \eta\ (([\,()\,], s), \mathit{Stack}\ xs') \\
\quad\quad\quad (\mathit{Left}\ r : xs') \quad \to h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND}+f} \circ (\Leftrightarrow)\ \$ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{do}\ \mathit{put}\ (\mathit{Stack}\ xs'); \mathit{restore}\ r; \mathit{untrail})\ s)\ t \\
= \quad \{\text{- case split -}\} \\
\quad h_{\mathit{State1}}\ (h_{\mathit{Modify1}}\ (h_{\mathit{ND}+f} \circ (\Leftrightarrow)\ \$ \\
\quad\quad \mathbf{do}\ \mathit{put}\ (\mathit{Stack}\ (ys' +\!\!+ (\mathit{Right}\ () : xs))); \mathit{restore}\ r; \mathit{untrail})\ s)\ t \\
= \quad \{\text{- definition of } h_{\mathit{State1}}, h_{\mathit{Modify1}}, h_{\mathit{ND}+f}, (\Leftrightarrow) \text{ and reformulation -}\} \\
\quad h_{\mathit{State1}}\ ((h_{\mathit{Modify1}} \circ h_{\mathit{ND}+f} \circ (\Leftrightarrow))\ \mathit{untrail}\ (s \ominus r)) \\
\quad\quad (\mathit{Stack}\ (ys' +\!\!+ (\mathit{Right}\ () : xs))) \\
= \quad \{\text{- induction hypothesis on } ys' \text{ -}\} \\
\quad \eta\ (([\,()\,], \mathit{fminus}\ (s \ominus r)\ ys'), \mathit{Stack}\ xs) \\
= \quad \{\text{- definition of } \mathit{fminus} \text{ -}\} \\
\quad \eta\ (([\,()\,], \mathit{fminus}\ s\ (\mathit{Left}\ r : ys')), \mathit{Stack}\ xs) \\
= \quad \{\text{- definition of } ys \text{ -}\} \\
\quad \eta\ (([\,()\,], \mathit{fminus}\ s\ ys), \mathit{Stack}\ xs)
\end{array}
$$

■

The following lemma is obvious from Lemma 18 and Lemma 19. It shows that we can restore the previous state and stack by pushing a time stamp on the trail stack and use the function *untrail* afterwards.

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press

Lemma 20 (State and stack are restored). *For* $t :: Stack\ (Either\ r\ ())$, $s :: s$, *and* $p ::$ $Free\ (Modify_F\ s\ r :+: Nondet_F :+: f)$ *a which does not use the restore operation, we have*

$$
\begin{aligned}
&\mathbf{do}\ ((\_, s_1), t_1) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (pushStack\ (Right\ ()))\ s)\ t \\
&\quad ((x, s_2), t_2) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s_1)\ t_1 \\
&\quad ((\_, s3), t3) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ untrail\ s_2)\ t_2 \\
&\quad \eta\ ((x, s3), t3) \\
=\ &\\
&\mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ (Stack\ [\,]) \\
&\quad \eta\ ((x, s), t)
\end{aligned}
$$

Proof Suppose $t = Stack\ xs$. We calculate as follows.

$$
\begin{aligned}
&\mathbf{do}\ ((\_, s_1), t_1) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (pushStack\ (Right\ ()))\ s)\ (Stack\ xs) \\
&\quad ((x, s_2), t_2) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s_1)\ t_1 \\
&\quad ((\_, s3), t3) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ untrail\ s_2)\ t_2 \\
&\quad \eta\ ((x, s3), t3) \\
=\ &\{\text{- definition of } h_{State1}, h_{Modify1}, h_{ND+f}, (\Leftrightarrow), pushStack \text{ -}\} \\
&\mathbf{do}\ ((\_, s_1), t_1) \leftarrow \eta\ (([()], s), Stack\ (Right\ () : xs)) \\
&\quad ((x, s_2), t_2) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s_1)\ t_1 \\
&\quad ((\_, s3), t3) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ untrail\ s_2)\ t_2 \\
&\quad \eta\ ((x, s3), t3) \\
=\ &\{\text{- monad law -}\} \\
&\mathbf{do}\ ((x, s_2), t_2) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s) \\
&\qquad\qquad (Stack\ (Right\ () : xs)) \\
&\quad ((\_, s3), t3) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ untrail\ s_2)\ t_2 \\
&\quad \eta\ ((x, s3), t3) \\
=\ &\{\text{- by Lemma 18, for some } ys = [Left\ r_1, ..., Left\ r_n] \text{ the equation holds -}\} \\
&\mathbf{do}\ ((x, s_2), t_2) \leftarrow \mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
&\qquad\qquad\qquad (local2trail\ p)\ s)\ (Stack\ [\,]) \\
&\qquad\qquad\qquad \eta\ ((x, fplus\ s\ ys), extend\ (Stack\ (Right\ () : xs))\ ys) \\
&\quad ((\_, s3), t3) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ untrail\ s_2)\ t_2 \\
&\quad \eta\ ((x, s3), t3) \\
=\ &\{\text{- monad law and definition of } extend \text{ -}\} \\
&\mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ (Stack\ [\,]) \\
&\quad ((\_, s3), t3) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow)) \\
&\qquad\qquad untrail\ (fplus\ s\ ys))\ (Stack\ (ys +\!\!+ (Right\ () : xs))) \\
&\quad \eta\ ((x, s3), t3) \\
=\ &\{\text{- Lemma 19 and monad law -}\} \\
&\mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ (Stack\ [\,]) \\
&\quad \eta\ ((x, fminus\ (fplus\ s\ ys)\ ys), Stack\ xs) \\
=\ &\{\text{- Equation (7.1) gives } fminus\ (fplus\ s\ ys)\ ys = s \text{ -}\} \\
&\mathbf{do}\ ((x, \_), \_) \leftarrow h_{State1}\ ((h_{Modify1} \circ h_{ND+f} \circ (\Leftrightarrow))\ (local2trail\ p)\ s)\ (Stack\ [\,]) \\
&\quad \eta\ ((x, s), Stack\ xs)
\end{aligned}
$$

■

https://doi.org/10.1017/S0956796824000133 Published online by Cambridge University Press