An Algebraic Effect for ML-Style References in Haskell

Daan Panis[1]

Supervisor(s): Casper Bach Poulsen[1]**, Jaro Reinders**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Daan Panis
Final project course: CSE3000 Research Project
Thesis committee: Casper Bach Poulsen, Jaro Reinders, Annibale Panichella

Abstract

Errors from side-effecting operations, such as mutable state, error handling, and I/O operations, can be costly during software development. Haskell's monadic approach often obscures specific operations, limiting the ability to reason about them effectively. This paper explores implementing ML-style references in Haskell using algebraic effects separating the syntax and semantics of side-effecting operations.

ML-style references are mutable storage locations, similar to pointers that ensure type safety and allow imperative programming within a functional language. We address how to implement ML-style references in Haskell using algebraic effects while adhering to Staton's state laws. Our contributions include developing an algebraic effect for mutable references, creating a corresponding handler, and proving adherence to Staton's state laws.

Additionally, we demonstrate the practical application of these principles by proving the correctness of an imperative-style factorial function. This work provides a flexible and predictable framework for using mutable references in Haskell, enhancing the ability to reason about program behaviour and correctness.

1 Introduction

Mistakes in software can be extremely expensive and significantly affect people's lives. Tools that help programmers reason about their programs and ensure their correctness can help reduce critical errors.

A common cause of mistakes is side-effecting operations such as mutable state, error handling, and I/O operations. While avoiding side-effecting operations entirely might be ideal, but their use is often necessary.

Haskell uses monads and monad transformers to manage side-effecting operations. However, a limitation of this approach is that most effects are represented through Haskell's IO monad, making it difficult to distinguish the exact side-effecting operations a program performs [2], making it harder to reason about them.

Algebraic effects, as introduced by Plotkin and Power [3], offer an alternative way to manage side effects. They separate the syntax and semantics of side-effecting operations from their implementation. This allows programmers to define different handlers for an effect and clearly see which side effects a program uses.

An interesting side effect is mutable references, as seen in ML [4]. Mutable references in ML are similar to pointers in C or Pascal but ensure type safety. Mutable references are well studied, Staton proposes laws governing mutable references [1], ensuring they are well-behaved and easy to reason about.

While mutable references and their corresponding state laws are well-studied, to our knowledge, no one has proven an effect handler for references that adheres to these state laws.

This paper addresses the following research question:

How can ML-style references [4] be implemented in Haskell using algebraic effects that adhere to global, block, and local state laws as defined by Staton [1]?

Answering this question enables programmers to write Haskell programs using mutable references in a predictable way. Staton's state laws allow for equational reasoning to prove properties of programs, ensuring their correctness.

Additionally, it enables the use of imperative constructs like while loops, which are common in languages like Java and C. This simplifies the creation of programs and data structures that are challenging to implement using standard Haskell techniques, such as doubly linked lists among many others.

Using algebraic effects for implementing mutable references abstracts away the handling of operations. This grants programmers greater control, allowing them to write handlers using different data structures for the store, tailored to their program's specific needs, making it a flexible way to use mutable references in Haskell.

This paper makes the following contributions:

1. Provides an algebraic effect in Haskell for implementing ML-style mutable references and a corresponding handler (Section 3, Section 4)
2. Proves that the proposed effect handler adheres to the global, block, and local state laws as defined by Staton (Section 5, Appendix A).
3. Demonstrates a clear and simple way to use Staton's state laws to reason about the correctness of programs using mutable references (Section 6)

The paper's structure is as follows: Section 2 covers necessary background on functors, monads, and algebraic effects using the Free monad. Section 3 describes mutable references in ML and proposes the operations for the algebraic effect. Section 4 provides an implementation for the $MLRef$ algebraic effect and a handler. Section 5 describes Staton's state laws and adapts them to Haskell syntax, with proofs in Appendix A. Section 6 demonstrates using the laws to prove the correctness of an imperative-style factorial program with the proposed algebraic effect. Section 7 discusses the ethical considerations of the research. Finally, Section 8 discusses the work and results of this paper, followed by the conclusion in Section 9.

2 Background

This paper assumes familiarity with Haskell concepts such as types, functions, and type classes. This section provides the necessary background information, beginning with an explanation of functors and monads. Next, the free monad is introduced, followed by an introduction to algebraic effects using the free monad.

2.1 Functors & Monads

A *Functor* is a type class that allows a function to be applied to each element within a structure without altering the structure itself. Here, "structure" refers to the organisation of the datatype.

The *Functor* type class in Haskell can be defined as follows:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Formally, a *Functor* must obey two laws: the identity law and the composition law. Although Haskell does not enforce these laws, it is the programmer's responsibility to ensure they are satisfied.

Identity	$fmap\ id \equiv id$
Composition	$fmap\ f \circ g \equiv fmap\ f \circ fmap\ g$

The identity law states that mapping the identity function over a *Functor* should return the *Functor* unchanged. The composition law asserts that mapping the composition of two functions over a *Functor* should be the same as first mapping one function and then the other.

A *Monad* is a type class that represents computations defined as a sequence of steps. It builds on the concept of a *Functor* by providing two primary operations, *return* and $\gg\!=$ (bind) [5]. The *Monad* type class can be defined as follows:

```
class Monad m where
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a
```

The *return* function injects a value into a monadic context making it a minimal monadic computation.

The $\gg\!=$ function, known as bind, is used to sequence monadic operations. It takes a monadic value and a function that returns a monadic value, chaining them together by passing the result of the first computation to the second.

Formally, a *Monad* must satisfy three laws: the left identity, the right identity, and associativity.

Left identity	$return\ a \gg\!= k \equiv k\ a$
Right identity	$m \gg\!= return \equiv m$
Associativity	$(m \gg\!= k) \gg\!= h \equiv m \gg\!= (\lambda x \rightarrow k\ x \gg\!= h)$

The left identity law states that wrapping a value with *return* and then applying a function using $\gg\!=$ is equivalent to applying the function directly. The right identity law asserts that applying $\gg\!=$ to a monadic value followed by *return* does not change the value. The associativity law ensures the order of monadic operations does not affect the result.

2.2 Free monad & Algebraic effects

The *Free* monad in Haskell represents computations as abstract syntax trees (ASTs). These trees are composed using a functor that defines the structure of the computation [6].

All code in this section is based on Bach Poulsen's blog post on algebraic effects [7], chosen for its simplicity and clear explanation. Similar approaches are described in other literature on the topic ([8, 6, 9, 10], among others).

```
data Free f a = Pure a | Op (f (Free f a))
```

The *Pure* constructor represents a completed computation with a value, while *Op* represents an ongoing computation involving an operation described by the functor f.

The *fold* function can be defined to recursively collapse the free monad. It interprets the free monad's abstract syntax tree into a concrete result by specifying how to handle both completed and ongoing computations.

```
fold :: Functor f ⇒
          (a → b) → (f b → b) → Free f a → b
fold gen _ (Pure x) = gen x
fold gen alg (Op f) = alg (fmap (fold gen alg) f)
```

Using this definition of *fold* and the *Free* monad data structure, we can effectively allow any functor to be a monad by representing it as a free monad. The *Monad* instance for *Free* is described below, the instances of *Functor* and *Applicative* are omitted here for brevity.

```
instance (Functor f) ⇒ Monad (Free f) where
  m >>= k = fold k Op m
  return = Pure
```

With this *Free* monad, we can start creating algebraic effects by describing them as *Functor* data types. For example, we can define the *Err* and *State* functors:

```
data Err k = Err String deriving Functor
data State s k = Put s k | Get (s → k)
  deriving Functor
```

These functors describe the operations available for the side effects. The *Err* side effect has a single operation that terminates the program with an error message, similar to exceptions in Java and C++. The *State* side effect has two operations: *Put*, which mutates the state, and *Get*, which retrieves the current state.

To use multiple algebraic effects in a single program, we represent them with a functor sum data type:

```
data (f + g) a = L (f a) | R (g a) deriving Functor
```

This datatype represents the sum of two functors f and g, meaning a value of $(f + g)\ a$ can either be type $f\ a$ or type $g\ a$ (wrapped in the L and R constructor).

This functor sum allows writing programs using multiple algebraic effects. For example, the following program reads the state, throws an error if the state is less than 10, and otherwise doubles the state value:

```
prog :: Free (State Int + Err) ()
prog = Op (L (Get (λn →
  if n < 10 then Op (R (Err "Less than 10"))
            else Op (L (Put (n * 2) (Pure ()))))))
```

However, writing programs this way is verbose because the L and R constructors must be manually specified. This also means that if we switch the order of the functors to *Free* (*Err* + *State Int*) (), we must update all L and R constructor invocations accordingly.

To address this issue, we use signature subtyping and smart constructors. Specifically, we define the subtyping $f < g$, which describes how to transform any $f\ a$ into a $g\ a$ [7].

```
class f < g where
  inj :: f k → g k
instance f < f where inj = id
instance f < (f + g) where inj = L
instance f < h ⇒ f < (g + h) where inj = R ∘ inj
```

The first instance uses the identity function for self-subtyping. The second uses L to inject f into the left of a sum type. The third uses R and recursion to inject f into the right of a sum type if f is a subtype of h.

Using signature subtyping, we use the following smart constructor definitions for *State* effect. The smart constructors inject the operation in the AST regardless of the order of the functor sum.

```
get :: State s < f ⇒ Free f s
get = Op (inj (Get Pure))
put :: State s < f ⇒ s → Free f ()
put s = Op (inj (Put s (Pure ())))
```

Using smart constructors, the *Monad* instance for *Free*, and Haskell's **do** notation [11], we can represent our previous program more plainly:

```
prog :: Free (State Int + Err) ()
prog = do
  n :: Int ← get
  if n < 10 then err "Less than 10"
               else put (n ∗ 2)
```

The *State* and *Err* effects only describe how they are used in a computation, not how they are implemented. To make them useful, we need a way to perform the effects. We start by using the following datatype:

```
data Handler f a f′ b = Handler
  {ret :: a → Free f′ b,
   hdlr :: f (Free f′ b) → Free f′ b}
```

The *ret* function of a handler defines how pure values of type a are handled, producing a result within the *Free* f' monad. The *hdlr* function defines how to handle operations of the effect f, such as implementing the *Err* effect. A handler typically handles a single side effect, leaving others intact. In a program with multiple effects, you must handle each effect sequentially to produce a final result. A handler for the *Err* effect could look like the following:

```
hErr :: (Functor f′) ⇒ Handler Err a f′
           (Either String a)
hErr = Handler
  {ret = pure ∘ Right,
    hdlr = λcase
      Err s → pure (Left s)}
```

This handler handles the *Err* operation, by transforming its result into an *Either String a*, meaning it can either be some error message or a result of a computation.

The *State* effect doesn't have any meaningful implementation using the *Handler* datatype, as it requires weaving some state through the computations. Instead, we use a different datatype:

```
data Handler_ f a p f′ b = Handler_
  {ret_ :: a → (p → Free f′ b),
   hdlr_ :: f (p → Free f′ b) → (p → Free f′ b)}
```

The difference is that *Handler_* has an additional type p, representing a value passed along during handling. Using this, we can write the following handler for the *State* effect:

```
hState :: Functor g ⇒ Handler_ (State s) a s g (a, s)
hState = Handler_
  {ret_ = λx s → pure (x, s)
  , hdlr_ = λx s → case x of
      Put s′ k → k s′
      Get k → k s s}
```

Here, *hState* handles the *State* effect with a state of type s. The result is a *Free* g with result type (a, s). The *Put* operation passes the newly set state to the continuation k, while the *Get* operation retrieves the state and passes it to the continuation s. We have omitted the functions that use these handlers to handle the effects for brevity, they are well described by Bach Poulsen [7].

One issue with this definition of handlers is that it requires a *Free* $f + f'$ as input, which cannot handle the last effect since it is not a sum of two functors. To solve this, you can introduce an empty data type:

```
data End k deriving Functor
```

Since this datatype is empty and has no operations, a *Free End* can only be a pure value. This way, *End* marks the end of the computation and contains the final result. In your programs, you always add the *End* effect to your effect sum. You can then retrieve the result of the computation after handling all effects using the *un* function:

```
un :: Free End a → a
un (Pure x) = x
un (Op f) = case f of     -- Can never happen
```

3 ML-style references

ML is a high-level functional programming language known for its strong typing and type inference, introduced by Milner [12]. It provides mechanisms for side-effecting operations, such as mutable references, enabling imperative-style code [13].

In ML, references are mutable storage locations similar to pointers in C and Pascal but designed to be secure and type-safe [13].

ML defines the following three operations for mutable references:

- The *ref* function creates a new reference. For example, *ref* 5 creates a reference to an integer with an initial value of 5.
- The ! operator is used to retrieve the value stored in a reference. For instance, $!p$ returns the value stored in the reference p.
- The := operator updates the value stored in a reference. For example, $p := 10$ sets the value of p to 10.

References enable imperative programming within ML, allowing programmers to write code that modifies state. This capability is crucial for implementing algorithms and data structures that require mutable references, such as doubly linked lists [13].

We propose a similar approach to references in ML by defining an algebraic effect in Haskell with three operations:

- An operation that allocates a new memory location in a store with a given value and returns a pointer to that location.
- An operation that retrieves the value at a memory location, given a pointer to that location.
- An operation that updates the value at a memory location, given a pointer to that location.

4 Implementation

With a clear understanding of the operations and method for implementing algebraic effects using the *Free* monad, we propose an algebraic effect for ML-style references. We define the data type for our operations, similar to the *State* effect, and propose a handler using a simple list as the store and an integer wrapper pointing to store locations.

4.1 Algebraic Effect

The data type *MLRef* parametrises over a reference type and a continuation type. The reference parameter can be any type constructor of kind $* \to *$, providing a flexible abstraction that allows for the creation of effect handlers with various reference types. The algebraic effect handler we implemented uses a specific reference type and state to manage the effect, as detailed later in this section.

```
data MLRef ref k where
  MkRef :: s → (ref s → k) → MLRef ref k
  DeRef :: ref s → (s → k) → MLRef ref k
  UpdateRef :: ref s → s → k → MLRef ref k

instance Functor (MLRef ref) where
  fmap f (MkRef v k) = MkRef v (λx → f (k x))
  fmap f (DeRef r k) = DeRef r (λx → f (k x))
  fmap f (UpdateRef r v k) = UpdateRef r v (f k)

mkref :: (MLRef ref < f) ⇒ a → Free f (ref a)
mkref v = Op (inj (MkRef v Pure))

deref :: (MLRef ref < f) ⇒ ref a → Free f a
deref r = Op (inj (DeRef r Pure))

update :: (Functor f, MLRef ref < f) ⇒
            ref a → a → Free f ()
update r v = Op (inj (UpdateRef r v (Pure ())))
```

Figure 1: *Implementation algebraic effect ML-Style references.*

As can be seen in the implementation in Figure 4.1, the *MLRef* data type defines three constructors:

- *MkRef*: This constructor creates a new reference. It takes a value of any type s and a function that receives a reference ($ref\ s$) and produces a continuation of type k. This continuation can be any computation involving the newly created reference.
- *DeRef*: This constructor is used to dereference a value. It takes a reference of any type $ref\ s$ and a function that receives a value of type s, producing a continuation of type k.
- *UpdateRef*: This constructor updates the value of an existing reference. It takes a reference of any type $ref\ s$, a new value of type s, and a continuation of type k to be executed after the update.

Furthermore, we introduced the smart constructors *mkref*, *deref*, and *update* to easily inject these operations into any program using the *MLRef* effect. With these operations defined, we can write programs using mutable references, as shown in the following example:

```
program :: Free (MLRef ref + End) Int
program = do
  r ← mkref 20
  v ← deref r
  update r (v * 10)
  deref r
```

Although this program is not particularly useful, it demonstrates how to write programs using the *MLRef* algebraic effect.

An alternative implementation of *MLRef* could involve parameterising s in the data type itself, rather than in each constructor, as shown below:

```
data MLRef ref s k   -- Constructors are unchanged
```

However, this approach requires manually specifying each type for which you want to create references. For example, if you want a program that can create references to both numbers and strings, you would need to specify:

```
program :: Free (MLRef ref Int +
                 MLRef ref String + End)
```

This quickly becomes unmanageable, especially when creating deeply nested references (e.g., a reference to a reference to a reference, etc.). You would need to manually add type annotations for each level of nesting, which becomes extremely inconvenient. Additionally, you would need to handle each reference type separately, potentially handling the program many times.

Our definition avoids this issue by not parameterising s in the data type definition, but only in the constructors. This makes handling slightly more complex, as it requires weakening the store to support any type of reference, which will be discussed in the next section.

4.2 Algebraic Effect Handler

The algebraic effect handler, shown in Figure 4.2, handles the *MLRef* effect using a specific reference type called *IntRef*. The *IntRef* type is a simple wrapper around an integer, which serves as an index into a list. The handler manages state using a list, where each *IntRef* acts as a pointer to an element within this list.

```
newtype IntRef a = IntRef Int deriving Functor

hRefCoerced :: (Functor g) ⇒
                 Handler_
                   (MLRef IntRef) a [p] g (a, [p])
hRefCoerced = Handler_
  {ret_ = curry pure,
   hdlr_ = λx state →
     let l = length state in case x of
       MkRef v k → k (IntRef l)
         (unsafeCoerce v : state)
       DeRef (IntRef i) k → k
         (unsafeCoerce
            (position (l − i − 1) state)) state
       UpdateRef (IntRef i) v k → k
         (replace (l − i − 1)
            (unsafeCoerce v) state)}
```

Figure 2: *Implementation algebraic effect handler ML-Style references for IntRef.*

This handler leverages Bach Poulsen's $\mathit{Handler}_{-}$ type, which provides a convenient mechanism to recursively handle all MLRef operations in a computation tree while leaving computations of other effect types unchanged.

The handler operates on the MLRef operations and some state as follows:

- MkRef: When handling the creation of a new reference, the handler prepends the provided value to the state list and creates an IntRef representing the index of this new value. The index corresponds to the length of the list prior to the insertion. The newly created IntRef is then passed to the continuation, along with the updated state.
- DeRef: Dereferencing retrieves the value from the state list at the position indicated by the IntRef. This value is then applied to the continuation, along with the unchanged state.
- $\mathit{UpdateRef}$: Updating a reference replaces the value in the state list at the position indicated by the IntRef with the new value. The updated state is then passed to the continuation.

Helper functions $\mathit{position}$ and $\mathit{replace}$ are used and provide list indexing and element replacement capabilities, respectively. Many other handlers operating on many other different types of references other than IntRef exist. The implementation provided here is a very simple one, but is much easier to prove compared to some alternatives, which is the reason for choosing this particular implementation.

5 Local & Global State Laws

Plotkin and Power first introduced the laws for global and local state in computational effects [14]. Staton expanded on these theories, offering a more comprehensive framework [1]. These laws describe interactions with mutable state: global state shared across the entire program, block state within a specific scope, and local state that can be dynamically created, updated, and read. To prove these laws hold for an algebraic effect handler for ML-style references, it is necessary to show that the handler accurately models these state interactions, maintaining the correct behaviour of stateful computations.

First, we modify Staton's syntax and definitions as needed for our proofs. Next, we outline the necessary assumptions for proving the laws, which we consider reasonable. Finally, we present the global, block, and local state laws using these modifications and then explain each law.

The proofs for the laws in this section, as applied to the proposed handler, can be found in Appendix A.

5.1 Modification to Staton's definitions

Staton's work describes the laws for global, block, and local state using Moggi's meta-language [15]. To prove these laws for our Haskell handler, we modify the language to match Haskell's syntax, simplifying the proofs and reasoning about programs using the algebraic effect.

We update Staton's syntax for reference operations to match our implementation: $\mathit{update}\ a\ v$ for $a := v$, $\mathit{deref}\ a$ for $!a$, and $\mathit{mkref}\ v$ for $\mathit{ref}\ v$. We replace Moggi's **let** notation with Haskell's **do** notation, requiring explicit return statements for pure values.

Consider Staton's first global state law:

$$a : \mathbb{A} \vdash \textbf{let}\ v \Leftarrow !a\ \textbf{in}\ a := v \equiv () : 1$$

By updating the reference operations and transforming to Haskell's **do** syntax, we get:

$$a : \mathbb{A} \vdash \textbf{do}\ v \leftarrow \mathit{deref}\ a\textbf{;}\ \mathit{update}\ a\ v \equiv \mathit{return}\ () : 1$$

These changes maintain consistency, preserving the operational semantics and sequential logic of the original laws in Haskell's do notation. Dereference and update operations, type safety, and return values remain equivalent, ensuring the correctness of the original proofs.

Staton describes $\mathbb{A}$ as an infinite set of atoms representing store locations [1]. Here, we define $\mathbb{A}$ as the set of IntRef instances with non-negative indices. This retains $\mathbb{A}$'s properties as an infinite collection of store references. Restricting to IntRef narrows the scope without altering the reference nature, preserving the algebraic theory's integrity as defined by Staton. We also restrict $\mathbb{V}$ to all valid Haskell values.

5.2 Assumptions

Since the implemented handler relies heavily on Haskell's $\mathit{unsafeCoerce}$ function to support a state list of any type, we must make the following two assumptions regarding the use of $\mathit{unsafeCoerce}$ in Haskell.

1. $v : \mathbb{V} \vdash \mathit{unsafeCoerce}\ v \equiv \mathit{unsafeCoerce}\ v$
2. $v : \mathbb{V} \vdash \mathit{unsafeCoerce}\ (\mathit{unsafeCoerce}\ v) \equiv v$

We deem these to be fair assumptions, particularly in the context of ML-style references, since they can be supported by the work done by Bach Poulsen et al. [16]. Their work emphasizes the importance of store extension and weakening environments, ensuring that type safety is maintained even when environments are extended. Additionally, the use of explicit weakening in the type of bind operations ensures that store extensions are well-handled, which aligns with the safe application of $\mathit{unsafeCoerce}$ in maintaining consistent type in this handler.

5.3 Global State

The theory of global state focuses on two operations: $lk : \mathbb{A} \rightarrow \mathbb{V}$ and $upd : \mathbb{A} \times \mathbb{V} \rightarrow 1$ [1]. In our context, these correspond to the update and deref operations. The global state laws, modified as previously defined, are shown in Figure 5.3.

GS1. $a : \mathbb{A} \vdash \textbf{do}\ v \leftarrow \mathit{deref}\ a\textbf{;}\ \mathit{update}\ a\ v$
$\equiv \mathit{return}\ () : 1$

GS2. $a : \mathbb{A} \vdash \textbf{do}\ v \leftarrow \mathit{deref}\ a\textbf{;}\ w \leftarrow \mathit{deref}\ a\textbf{;}$
$\mathit{return}\ (v, w) \equiv \textbf{do}\ v \leftarrow \mathit{deref}\ a\textbf{;}$
$\mathit{return}\ (v, v) : \mathbb{V} \times \mathbb{V}$

GS3. $a : \mathbb{A}, v, w : \mathbb{V} \vdash \textbf{do}\ \mathit{update}\ a\ v\textbf{;}\ \mathit{update}\ a\ w\textbf{;}$
$\equiv \textbf{do}\ \mathit{update}\ a\ w : 1$

GS4. $a : \mathbb{A}, v : \mathbb{V} \vdash \textbf{do}\ \mathit{update}\ a\ v\textbf{;}\ w \leftarrow \mathit{deref}\ a\textbf{;}$
$\textbf{do}\ \mathit{update}\ a\ v\textbf{;}\ \mathit{return}\ v : \mathbb{V}$

GS5. $(a, b) : \mathbb{A} \otimes \mathbb{A} \vdash \textbf{do}\ v \leftarrow \mathit{deref}\ a\textbf{;}\ w \leftarrow \mathit{deref}\ b\textbf{;}$
$\mathit{return}\ (v, w) \equiv \textbf{do}\ w \leftarrow \mathit{deref}\ b\textbf{;}$
$v \leftarrow \mathit{deref}\ a\textbf{;}\ \mathit{return}\ (v, w) : \mathbb{V} \times \mathbb{V}$

GS6. $(a, b) : \mathbb{A} \otimes \mathbb{A}, v, w : \mathbb{V} \vdash \textbf{do}\ \mathit{update}\ a\ v\textbf{;}$
$\mathit{update}\ b\ w \equiv \textbf{do}\ \mathit{update}\ b\ w\textbf{;}\ \mathit{update}\ a\ v : 1$

GS7. $(a, b) : \mathbb{A} \otimes \mathbb{A}, v : \mathbb{V} \vdash \textbf{do}\ \mathit{update}\ a\ v\textbf{;}\ \mathit{deref}\ b$
$\textbf{do}\ w \leftarrow \mathit{deref}\ b\textbf{;}\ \mathit{update}\ a\ v\textbf{;}\ \mathit{return}\ w : \mathbb{V}$

Figure 3: *Global state laws as described Staton, adapted to use the modified syntax.*

The global state laws govern the behaviour of mutable references and their interactions. These laws ensure consistency in how state is read and updated within a program.

- GS1 states that reading and then writing the same value back to a reference is equivalent to doing nothing.
- GS2 states that reading a value from a reference twice will yield the same value both times.
- GS3 states that updating a reference twice in succession is equivalent to just performing the second update.
- GS4 states that after updating a reference, reading from it will yield the updated value.
- GS5 states that the order of reading values from two distinct references does not matter.
- GS6 states that updating two distinct references in any order results in the same final state.
- GS7 states that updating one reference and reading from another can be reordered without changing the result.

5.4 Theory of Block

Staton describes the disjoint product $\mathbb{A}^{\otimes n}$ as an n-tuple of distinct atoms, meaning n distinct references pointing to different locations in the store [1].

They define $\mathit{ref}^n : (\mathbb{A}^{\otimes n} \times \mathbb{V}) \rightarrow \mathbb{A}^{\otimes (n+1)}$, which we rename as $\mathit{mkref}^n\ (\vec{a}, v)$, to allocate a new reference distinct from all in $\vec{a}$.

Finally, the function p_n, an injection $\mathbb{A}^{\otimes (n+1)} \rightarrow \mathbb{A}^{\otimes n} \times \mathbb{A}$, returns the first n locations from a set of $n+1$ locations, separating the first n elements from the last one.

The block state laws, using the previously stated modifications, can be seen in Figure 5.4.

B1. $v : \mathbb{V} \vdash \textbf{do}\ a \leftarrow \mathit{mkref}\ v\textbf{;}\ \mathit{return}\ () \equiv \mathit{return}\ () : 1$

B2. $v, w : \mathbb{V} \vdash \textbf{do}\ a \leftarrow \mathit{mkref}\ v\textbf{;}\ b \leftarrow \mathit{mkref}\ w\textbf{;}$
$\mathit{return}\ (a, b) \equiv \textbf{do}\ b \leftarrow \mathit{mkref}\ w\textbf{;}$
$a \leftarrow \mathit{mkref}\ v\textbf{;}\ \mathit{return}\ (a, b) : \mathbb{A} \times \mathbb{A}$

B3$_n$. $v : \mathbb{V}, \vec{a} : \mathbb{A}^{\otimes n} \vdash \textbf{do}\ \vec{b} \leftarrow \mathit{mkref}^n(\vec{a}, v)\textbf{;}$
$\mathit{return}\ p_n(\vec{b}) \equiv \textbf{do}\ b \leftarrow \mathit{mkref}\ v\textbf{;}$
$\mathit{return}\ (\vec{a}, b) : \mathbb{A}^{\otimes n} \times \mathbb{A}$

Figure 4: *Block state laws as described Staton, adapted to use the modified syntax.*

The block state laws ensure consistent behaviour for allocating references within a specific scope.

- B1 states that creating a reference and immediately returning is equivalent to doing nothing.
- B2 states that the order of creating two references directly after each other does not matter.
- B3$_n$ states that allocating a new reference distinct from an existing set of references is equivalent to independently allocating a new reference and adding it to the original set. Its purpose is to ensure that each new reference that is created is unique.

Law B3$_n$ might seem confusing without a definition of the mkref^n function for the handler. However, for the implementation, we argue that B3$_n$ is trivially correct because the implementation of the mkref operation always creates a unique reference pointing to a distinct memory location from all other allocated references. Further details are provided in Section A.4.

5.5 Local State

Staton introduces local state as the combination of the theory of global state with the theory of block, achieved by providing four additional laws using a combination of the mkref, deref, and update operations [1]. Figure 5.5 presents these additional laws with our syntax modifications.

LS1. $v : \mathbb{V} \vdash \textbf{do}\ a \leftarrow \mathit{mkref}\ v\textbf{;}\ \mathit{update}\ a\ w\textbf{;}$
$\mathit{return}\ a \equiv \textbf{do}\ a \leftarrow \mathit{mkref}\ w\textbf{;}\ \mathit{return}\ a : \mathbb{A}$

LS2. $v : \mathbb{V} \vdash \textbf{do}\ a \leftarrow \mathit{mkref}\ v\textbf{;}\ w \leftarrow \mathit{deref}\ a\textbf{;}$
$\mathit{return}\ (w, a) \equiv \textbf{do}\ a \leftarrow \mathit{mkref}\ v\textbf{;}$
$\mathit{return}\ (v, a) : \mathbb{V} \times \mathbb{A}$

LS3. $a : \mathbb{A}, v, w : \mathbb{V} \vdash \textbf{do}\ b \leftarrow \mathit{mkref}\ v\textbf{;}\ \mathit{update}\ a\ w\textbf{;}$
$\mathit{return}\ b \equiv \textbf{do}\ \mathit{update}\ a\ w\textbf{;}\ b \leftarrow \mathit{mkref}\ v\textbf{;}$
$\mathit{return}\ b : \mathbb{A}$

LS4. $a : \mathbb{A}, v : \mathbb{V} \vdash \textbf{do}\ b \leftarrow \mathit{mkref}\ v\textbf{;}\ w \leftarrow \mathit{deref}\ a$
$\mathit{return}\ (w, b) \equiv \textbf{do}\ w \leftarrow \mathit{deref}\ a\textbf{;}$
$b \leftarrow \mathit{mkref}\ v\textbf{;}\ \mathit{return}\ (w, b) : \mathbb{V} \times \mathbb{A}$

Figure 5: *Local state laws as described Staton, adapted to use the modified syntax.*

The local state laws ensure consistent behaviour for dynamically allocated references that can be read an updated.

- LS1 states that creating a reference and immediately updating it is equivalent to creating the reference with the updated value.
- LS2 states that creating a reference, reading its value, and returning a pair of the reference and the value is the same as creating the reference and immediately returning the reference and the value without reading it first.
- LS3 states that the order of creating a new reference and updating a different reference does not matter.
- LS4 states that the order of creating a new reference and reading the value of a different reference does not matter.

6 Proving correctness of programs

Having a handler that complies with the state laws provides a powerful tool for reasoning about programs that use mutable references. These laws enable you to use equational reasoning to prove the correctness of programs in a straightforward and intuitive manner. We demonstrate its usefulness by proving the correctness of a Haskell program that computes the factorial in an imperative style using the *MLRef* effect, as shown in Figure 6.

```
while :: (Functor f) ⇒
           Free f Bool → Free f () → Free f ()
while cond body = do
  c ← cond
```

```
    if $c$ then do $body$; $while\ cond\ body$
        else $return$ ()
$impFact$ :: $Int \to Free\ (MLRef\ ref + End)\ Int$
$impFact\ n$ = do
  $x \leftarrow mkref\ 1$; $acc \leftarrow mkref\ 1$
  $while$
    ($deref\ x \gg\!= return \circ (\leqslant n)$)
    (do $x' \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
          $update\ acc\ (acc' * x')$; $update\ x\ (x' + 1)$
    )
  $deref\ acc$
```

Figure 6: *Factorial function, in an imperative style using a custom while construct, using the reference algebraic effect.*

We introduce a $while$ function that takes two parameters: a condition and a body. The condition should be a computation that checks whether the while loop should continue. The body should be a computation that performs an action, usually updating some state to ensure the while loop will eventually terminate.

The $impFact$ function uses the while loop to iterate until the reference x exceeds n. In the body, it multiplies the accumulator reference acc by x and then increments x by one.

We want to prove that the $impFact\ n$ function correctly computes the factorial of n, as defined as:

$$0! = 1, \qquad n! = 1 \cdot 2 \cdots (n-1) \cdot n$$

We will use the state laws to prove the properties of this function. In the proof, we annotate the laws used at each step, although we skip over annotating most order of operations laws. We freely change the order of operations that clearly do not conflict with each other, such as using different references or dereferencing the same value multiple times without intermediate updates.

Proof. We will start by analysing and reasoning about the $while$ loop in the program. The definition of the $while$ function, when expanded with our definition of $cond$, looks like this:

```
$xVal \leftarrow deref\ x$
if $xVal \leqslant n$ then $body$; $while\ cond\ body$
              else $return$ ()
```

First, consider the case where $xVal > n$. Here, the loop terminates with $return$ (). By GS1, we know that $return$ () is equivalent to dereferencing a reference and then updating it with the previously dereferenced value:

```
$xVal \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
$update\ acc\ acc'$
```

Now, consider the case where $xVal \leqslant n$. Expanding the $body$ definition in this context, we obtain:

```
$xVal \leftarrow deref\ x$
$x' \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
$update\ acc\ (acc' * x')$; $update\ x\ (x' + 1)$
$while\ cond\ body$
```

Applying GS2, which tells us that dereferencing the same reference multiple times in succession is redundant, we can remove the second dereference of x and replace x' with $xVal$:

```
$xVal \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
$update\ acc\ (acc' * xVal)$; $update\ x\ (xVal + 1)$
$while\ cond\ body$
```

Next, expanding the $while$ loop again with our definition of $cond$, we get:

```
$xVal \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
$update\ acc\ (acc' * xVal)$; $update\ x\ (xVal + 1)$
$xVal' \leftarrow deref\ x$
if $xVal' \leqslant n$ then $body$; $while\ cond\ body$
              else $return$ ()
```

Using GS4, which states that updating a reference and then immediately dereferencing it is redundant, we can omit $xVal' \leftarrow deref\ x$ and substitute its occurrences with $xVal + 1$. In the case where $xVal + 1 \leqslant n$, we expand the body again:

```
$xVal \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
$update\ acc\ (acc' * xVal)$; $update\ x\ (xVal + 1)$
$acc'' \leftarrow deref\ acc$
$update\ acc\ (acc'' * (xVal + 1))$; $update\ x\ (xVal + 2)$
$while\ cond\ body$
```

By applying GS4 again and using GS3, which tells us that updating a reference twice (or more) in a row is redundant, we simplify the sequence:

```
$xVal \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
$update\ acc\ (acc' * xVal * (xVal + 1))$
$update\ x\ (xVal + 2)$
$while\ cond\ body$
```

Continuing this pattern, we observe that each iteration multiplies acc by successive integers up to n and increments x. Therefore, the $while$ loop of $impFact$ can be summarised as:

```
$xVal \leftarrow deref\ x'$; $acc' \leftarrow deref\ acc$
$update\ acc\ (acc' * xVal * (xVal + 1) * ... * n)$
$update\ x\ n$
```

Substituting this back into the $impFact$ function, we get:

```
$x \leftarrow mkref\ 1$; $acc \leftarrow mkref\ 1$
$xVal \leftarrow deref\ x$; $acc' \leftarrow deref\ acc$
$update\ acc\ (acc' * xVal * (xVal + 1) * ... * n)$
$update\ x\ n$
$deref\ acc$
```

Applying LS2, which tells us that dereferencing right after creating the reference is redundant, and LS1, which states that creating a reference and then immediately updating it is equivalent to just making a reference with the updated value, we simplify the creation and immediate updating of references:

```
$x \leftarrow mkref\ 1$; $acc \leftarrow mkref\ 1$
$update\ acc\ (1 * 1 * 2 * ... * n)$; $update\ x\ n$
$deref\ acc$
```

Finally, by using B1, which states that creating a reference and then immediately dereferencing it is unnecessary, we can further simplify:

```
$return\ (1 * 1 * 2 * ... * n)$
```

This final expression is clearly equivalent to the definition of the factorial function, confirming that the program correctly computes the factorial of n. □

This proof provides a clear procedure for proving the correctness of programs using mutable references that adhere to state laws. This approach is appealing because it allows you to develop applications with mutable references, execute them, and reason about their correctness entirely within Haskell, without needing different models or environments for program verification.

7 Responsible Research

This work is designed to be easily reproducible using the source code provided in a GitHub repository. [17]. The source code in the repository is made available under the MIT License, ensuring that anyone can run, modify, and distribute the code freely. The repository contains documentation, providing instructions on how to execute the code and explanations of each component's functionality. This facilitates other researchers in reproducing our work and building upon it.

Additionally, all theorems and their corresponding proofs are included both in this paper and within the repository. This thorough documentation ensures that all theoretical results can be independently verified and correlated with the implementation.

These things together allow any researcher to reproduce, verify, and build on top of the work provided in this paper.

8 Discussion

In this chapter, we discuss the results and findings of our research. We begin by discussing related work and how our work builds upon it, followed by the limitations of our proposed handler. Finally, we explore potential optimisation strategies using the state laws.

8.1 Related work

This research builds upon several key works in algebraic effects and mutable references. Staton's framework on global, block, and local state laws provides the theoretical basis for reasoning about mutable references [1]. Our work extends these concepts by implementing an effect handler for ML-style references in Haskell, proving its adherence to the state laws.

We draw on the principles of algebraic effects introduced by Plotkin and Power [3], which separate the definition of side-effecting operations from their implementation, allowing for more flexible and clear reasoning about programs. We use Bach Poulsen's blog post on algebraic effects in Haskell as a framework for developing our proposed algebraic effect [7]. Additionally, we use the work by Bach Poulsen et al. on type correctness for languages using ML-style references and methods for handling store and value weakening, which supports the practical aspects of our implementation [16].

Our research demonstrates how to implement ML-style references in Haskell and shows a method for using algebraic effects to verify the correctness of programs that use mutable references. This combination of theoretical and practical insights highlights the potential of algebraic effects to simplify and improve the development of reliable software.

8.2 Limitations

The provided implementation of an ML-style reference algebraic effect demonstrates its usefulness but also has some limitations.

The primary limitation is performance. We used a basic implementation using a list of values to represent the store and the IntRef data type to represent pointers. While this approach is easy to reason about, it sacrifices performance. Creating new references is fast, as you can simply insert at the start of the list. However, the update and read methods have a worst-case time complexity of $\mathcal{O}(n)$, making them inefficient compared to other data structures.

Using $\mathit{unsafeCoerce}$ allowed for a flexible handler that supports any type of value in the store. However, this flexibility comes at the cost of losing type information about the store after handling a program. As a result, inspecting the store becomes nearly impossible because $\mathit{unsafeCoerce}$ causes the type to be indeterminate. Alternatively, we could have used Haskell's $\mathit{Dynamic}$ type, which allows conversion of any $\mathit{Typeable}$ instance into a $\mathit{Dynamic}$ and vice versa. This would enable type-safe inspection but would limit the values used to create references to those that are $\mathit{Typeable}$.

8.3 Optimisations

An interesting topic of discussion is how the state laws can optimise programs using the algebraic effect. A basic optimisation strategy involves using these laws to remove redundant operations. For example, dereferencing the same reference twice without an intermediate update is redundant. We can optimise the code to dereference only once and use that value directly. This strategy is relatively easy to implement and can be performed at compile time.

Section 6 reveals a more complex strategy. We demonstrate using the laws to transform an imperative-style factorial program into its recursive definition. This suggests it is possible to rewrite some imperative programs into functional recursive versions without mutable references automatically. While this may not apply to all programs, it could work for many.

However, the desirability of such transformations is debatable. In some cases, using mutable references (especially those not requiring $\mathcal{O}(n)$ time to dereference) might be faster. For a simple factorial function, the functional version is likely more beneficial, as Haskell is highly optimised for such recursive functions. Nonetheless, there will be situations where using mutable references could be faster, even if the optimiser can rewrite them.

9 Conclusion

This research aimed to address how ML-style references can be implemented in Haskell using algebraic effects that comply with state laws as described by Staton.

We introduced an algebraic effect in Haskell that represents operations similar to references in ML, along with a basic implementation of a handler adhering to these state laws. We provided proofs demonstrating that the effect handler complies with all the described state laws.

The practical usability of this algebraic effect was shown by proving the correctness of an imperative-style factorial function in Haskell, illustrating that the algebraic effect and the corresponding state laws can be used to reason about and verify the correctness of programs that use the effect, in a straightforward manner.

By exploring these points, we addressed the main research question. We implemented an effect and handler that comply with the state laws as described by Staton and demonstrated a clear method for using these laws to prove the correctness of programs using the effect. This contribution enables users to write programs using ML-style references, reason about their properties and correctness, and execute them directly in Haskell.

Overall, this paper provides programmers with a clean and efficient way to write, run, and reason about programs with mutable references in Haskell, suggesting potential further research and improvement in this area.

9.1 Recommendations for future work

We noted that the proposed handler lacks performance efficiency. Exploring more efficient implementations using data structures other than lists would be beneficial. Alternatives such as Haskell's *IORef* or *STRef* for mutable references could be considered, though proving the local and global state laws for a handler using these may be more challenging. However, these laws are likely to hold in practice.

A promising area for future research is examining how the proposed effect interacts with other common effects, such as state, exceptions, and nondeterminism. Understanding these interactions can enhance the usefulness of this algebraic effect in larger systems using multiple effects, thereby increasing its real-world applicability. This knowledge can also help with reasoning about programs with multiple effects, which is a major motivation for this paper. Additionally, it could lead to optimisations that improve the performance and efficiency of programs using combined effects.

Finally, we recommend further research into optimisers for programs using the algebraic effect. As described in Section 8, there are strategies to optimise such programs using the local and global state laws. Reducing the overhead of the algebraic effect and optimising user-written programs could significantly enhance its real-world applicability, making it competitive with other alternatives.

References

[1] Staton, S., "Completeness for Algebraic Theories of Local State," *Foundations of Software Science and Computational Structures*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 48–63.

[2] Swierstra, W., "Data types à la carte," *Journal of Functional Programming*, Vol. 18, No. 4, 2008, p. 423–436.

[3] Plotkin, G., and Power, J., "Adequacy for Algebraic Effects," *Foundations of Software Science and Computation Structures*, Vol. 2030, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 1–24.

[4] Milner, R., Harper, R., MacQueen, D., and Tofte, M., *The Definition of Standard ML*, The MIT Press, 1997.

[5] Wadler, P., "The essence of functional programming," *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92*, ACM Press, Albuquerque, New Mexico, United States, 1992, pp. 1–14.

[6] Yang, Z., and Wu, N., "Reasoning about effect interaction by fusion," *Proceedings of the ACM on Programming Languages*, Vol. 5, No. ICFP, 2021, pp. 1–29.

[7] Bach Poulsen, C., "Algebraic Effects and Handlers in Haskell," , Jul. 2023. URL `http://casperbp.net/posts/2023-07-algebraic-effects/`.

[8] Wu, N., and Schrijvers, T., "Fusion for Free: Efficient Algebraic Effect Handlers," *Mathematics of Program Construction*, Vol. 9129, edited by R. Hinze and J. Voigtländer, Springer International Publishing, Cham, 2015, pp. 302–322.

[9] Pretnar, M., "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper," *Electronic Notes in Theoretical Computer Science*, Vol. 319, 2015, pp. 19–35.

[10] Leijen, D., "Algebraic Effects for Functional Programming," , August 2016. URL `https://www.microsoft.com/en-us/research/publication/algebraic-effects-for-functional-programming/`.

[11] Marlow, S., "Haskell 2010 Language Report," , 2010. URL `https://cse.sc.edu/~mgv/csce330f16/haskell/haskell2010.pdf`.

[12] Milner, R., "A proposal for standard ML," *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, Association for Computing Machinery, New York, NY, USA, 1984, p. 184–197.

[13] Paulson, L. C., *ML for the working programmer*, 2nd ed., University of Cambridge, 1996.

[14] Plotkin, G., and Power, J., "Notions of Computation Determine Monads," *Foundations of Software Science and Computation Structures*, Vol. 2303, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 342–356.

[15] Moggi, E., "Notions of computation and monads," *Information and Computation*, Vol. 93, No. 1, 1991, pp. 55–92.

[16] Bach Poulsen, C., Rouvoet, A., Tolmach, A., Krebbers, R., and Visser, E., "Intrinsically-typed definitional interpreters for imperative languages," *Proceedings of the ACM on Programming Languages*, Vol. 2, No. POPL, 2018, pp. 1–34.

[17] Panis, D., "Algebraic Effect for ML-Style References in Haskell," `https://github.com/daanpanis/MLRef`, 2024.

A Proofs: Staton's State Laws

In this section, we prove the state laws for mutable references as defined by Staton. We begin by describing several theorems and corollaries, along with their corresponding proofs, related to our specific handler. We then propose and prove theorems concerning the position and replace helper functions. Finally, we prove the global, block, and local state laws for the proposed handler.

A.1 Effect handler

We begin by defining *hFold*, which will be used to simplify our proofs, as this expression will appear frequently.

Definition 1. *We define the following function in order to be able to shorten the proofs. This code will be used and substituted freely within the proofs.*

```
hFold = fold
  (ret_ hRefCoerced)
  (λcase
    L x → hdlr_ hRefCoerced x
    R x → λp → Op (fmap (λm → m p) x))
```

Next, we propose a theorem regarding the behaviour of the *handle_* function when handling *MLRef* operations.

Theorem 1. *Let op be any of the described operations of the MLRef algebraic effect.*

```
hFold (Op (L op)) state
≡
(λx state →
  let l = length state
    in case x of
      MkRef v k →
        k (IntRef l)
          (unsafeCoerce v : state)
      DeRef (IntRef i) k →
        k (unsafeCoerce (position (l − i − 1) state))
          state
      UpdateRef (IntRef i) v k →
        k
          (replace (l − i − 1)
            (unsafeCoerce v) state))
(fmap hFold op)
state
```

Proof.

```
hFold (Op (L op)) state
 ≡   -- By definition of hFold and the fold function.
(λcase
  L x → hdlr_ hRefCoerced x
  R x → λp → Op (fmap (λm → m p) x))
(fmap hFold (L op))
state
 ≡   -- definition of Functor (f + g),
       -- fmap (L x) is equal to L (fmap x)
(λcase
  L x → hdlr_ hRefCoerced x
  R x → λp → Op (fmap (λm → m p) x))
(L (fmap hFold op))
state
 ≡   -- Apply (L x) case
hdlr_
  hRefCoerced
  (fmap hFold op)
  state
 ≡   -- By definition of hdlr_ hRefCoerced
(λx state →
  let l = length state
  in case x of
    MkRef v k →
      k (IntRef l)
        (unsafeCoerce v : state)
    DeRef (IntRef i) k →
      k (unsafeCoerce (position (l − i − 1) state))
        state
    UpdateRef (IntRef i) v k →
    k
      (replace (l − i − 1)
        (unsafeCoerce v) state))
(fmap hFold op)
state
```

□

Corollary 1.1.

```
hFold (Op (L (MkRef v k))) state
 ≡
hFold (k (IntRef l)) (unsafeCoerce v : state)
```

Proof.

```
hFold (Op (L (MkRef v k))) state
 ≡   -- By Theorem 1
(λx state →
  let l = length state
  in case x of
    MkRef v k →
      k (IntRef l)
        (unsafeCoerce v : state)
    DeRef (IntRef i) k →
      k (unsafeCoerce (position (l − i − 1) state))
        state
    UpdateRef (IntRef i) v k →
      k
        (replace (l − i − 1)
          (unsafeCoerce v) state))
(fmap hFold (MkRef v k)))
state
 ≡   -- By definition of Functor (MLRef ref)
(λx state →
  let l = length state
  in case x of
    MkRef v k →
      k (IntRef l)
        (unsafeCoerce v : state)
    DeRef (IntRef i) k →
      k (unsafeCoerce (position (l − i − 1) state))
        state
    UpdateRef (IntRef i) v k →
      k
        (replace (l − i − 1)
          (unsafeCoerce v) state))
```

```
(MkRef v (λx → hFold (k x)))
state
≡   -- Apply the operation and the state to the lambda
(λx → hFold (k x)) (IntRef l) (unsafeCoerce v : state)
≡   -- Substitute in lambda
hFold (k (IntRef l)) (unsafeCoerce v : state)
```

□

Corollary 1.2.

```
hFold (Op (L (DeRef r k))) state
≡
hFold (k (unsafeCoerce
  (position (l − i − 1) state)))
  state
```

Proof.

```
hFold (Op (L (DeRef r k))) state
≡   -- By Theorem 1
(λx state →
        let l = length state
        in case x of
          MkRef v k →
            k (IntRef l)
              (unsafeCoerce v : state)
          DeRef (IntRef i) k →
            k (unsafeCoerce (position (l − i − 1) state))
              state
          UpdateRef (IntRef i) v k →
            k
              (replace (l − i − 1)
                (unsafeCoerce v) state))
(fmap hFold (DeRef v k)))
state
≡   -- By definition of ‘Functor MLRef‘
(λx state →
    let l = length state
      in case x of
        MkRef v k →
            k (IntRef l)
              (unsafeCoerce v : state)
        DeRef (IntRef i) k →
            k (unsafeCoerce (position (l − i − 1) state))
              state
        UpdateRef (IntRef i) v k → k
            (replace (l − i − 1) (unsafeCoerce v) state))
(DeRef r (λx → hFold (k x)))
state
≡   -- Apply the operation and the state to the lambda
(λx → hFold (k x))
        (unsafeCoerce (position (l − i − 1) state))
          state
≡   -- Apply lambda
hFold (k (unsafeCoerce (position (l − i − 1) state)))
        state
```

□

Corollary 1.3.

```
hFold
  (Op (L (UpdateRef r v k)))
  state
≡
(hFold k) (replace (l − i − 1) (unsafeCoerce v) state)
```

Proof.

```
hFold
  (Op (L (UpdateRef r v k)))
  state
≡   -- By Theorem 1
(λx state →
  let l = length state
    in case x of
      MkRef v k →
        k (IntRef l)
          (unsafeCoerce v : state)
      DeRef (IntRef i) k →
        k (unsafeCoerce (position (l − i − 1) state))
          state
      UpdateRef (IntRef i) v k → k
        (replace (l − i − 1)
          (unsafeCoerce v) state))
(fmap hFold (UpdateRef r v k)))
state
≡   -- By definition of ‘Functor MLRef‘
(λx state →
  let l = length state
    in case x of
      MkRef v k →
        k (IntRef l)
          (unsafeCoerce v : state)
      DeRef (IntRef i) k →
        k (unsafeCoerce (position (l − i − 1) state))
        state
      UpdateRef (IntRef i) v k →
        k (replace (l − i − 1) (unsafeCoerce v) state))
(UpdateRef r v (hFold k))
state
≡   -- Apply the operation and the state to the lambda
(hFold k) (replace (l − i − 1) (unsafeCoerce v) state)
```

□

A.2 Replace & Position

In this section, we will state various theorems and proofs for the replace and position helper functions. These theorems are crucial for proving Staton's state laws, as will be demonstrated in the following chapters.

Theorem 2. *Given any state such that* $\mathit{length}\ \mathit{state} > 0$*, and given* $0 \leqslant i < \mathit{length}\ \mathit{state}$*, it holds that:*

$$\mathit{replace}\ i\ (\mathit{position}\ i\ \mathit{state})\ \mathit{state} \equiv \mathit{state}$$

Proof. To prove that replacing the value at position i in *state*, with the value at position i in *state*, we use induction on i.

Base case: For the base case we let $i = 0$. We know that *state* is not empty:

```
replace 0 (position 0 (x : xs)) (x : xs)
 ≡   -- By definition of ‘replace‘
(position 0 (x : xs)) : xs
```

$\equiv$ -- By definition of 'position'
$x : xs$

Inductive step: Assume the theorem holds for $i = k$, for some $k \geqslant 0$. We prove the statement holds for $i = k + 1$:

$\mathit{replace}\ (k+1)\ (\mathit{position}\ (k+1)$
$(x : xs))\ (x : xs)$
$\equiv$ -- By definition of $\mathit{position}$ when $n > 0$
$\mathit{replace}\ (k+1)\ (\mathit{position}\ k\ xs)\ (x : xs)$
$\equiv$ -- By definition of $\mathit{replace}$ when $n > 0$
$x : \mathit{replace}\ k\ (\mathit{position}\ k\ xs)$
$\equiv$ -- By the inductive hypothesis
$x : xs$

□

Theorem 3. *Given any state such that length state > 0, given $0 \leqslant i <$ length state, and given any $v, w \in \mathbb{V}$*

$$\mathit{replace}\ (i\ w\ (\mathit{replace}\ i\ v\ \mathit{state})) \equiv \mathit{replace}\ i\ w\ \mathit{state}$$

Proof. To prove the prove that replacing the value of a reference twice sequentially is the same as only performing the replace, we use induction on i.

Base case: For the base case we let $i = 0$, and we know state is not empty.

$\mathit{replace}\ 0\ (\mathit{position}\ 0\ (x : xs))\ (x : xs)$
$\equiv$ -- By definition of $\mathit{replace}$
$(\mathit{position}\ 0\ (x : xs)) : xs$
$\equiv$ -- By definition of $\mathit{position}$
$x : xs$

Inductive step: Assume the statement holds for $i = k$, where $k > 0$ is an arbitrary number. We prove that the statement also holds for $i = k + 1$.

$\mathit{replace}\ (k+1)\ (\mathit{position}\ (k+1)\ (x : xs))\ (x : xs)$
$\equiv$ -- By definition of $\mathit{position}$ when $n > 0$
$\mathit{replace}\ (k+1)\ (\mathit{position}\ k\ xs)\ (x : xs)$
$\equiv$ -- By definition of $\mathit{replace}$ when $n > 0$
$x : \mathit{replace}\ k\ (\mathit{position}\ k\ xs)$
$\equiv$ -- By the inductive hypothesis
$x : xs$

□

Theorem 4. *Given any state such that length state >0. Given $0 \leqslant i, j <$ length state such that $i \not\equiv j$. For any $v \in \mathbb{V}$ it holds that:*

$$\mathit{position}\ i\ (\mathit{replace}\ i\ v\ \mathit{state}) \equiv v$$

Proof. To prove that retrieving a value at position i after replacing a value at position i with value v, is the same as just the value v, we use induction on i.

Base case: For the base case we let $i = 0$. We know state is not empty.

$\mathit{position}\ 0\ (\mathit{replace}\ 0\ v\ (x : xs)) = v$
$\equiv$ -- By definition of $\mathit{replace}$, for $n = 0$
$\mathit{position}\ 0\ (v : xs)$
$\equiv$ -- By definition of $\mathit{position}$, for $n = 0$
v

Inductive step: Assume the statement holds for $i = k$, where $k > 0$ is an arbitrary number. We prove that the statement also holds for $i = k + 1$.

$\mathit{position}\ (k+1)\ (\mathit{replace}\ (k+1)\ v\ (x : xs))$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$
$\mathit{position}\ (k+1)\ (x : \mathit{replace}\ k\ v\ xs)$
$\equiv$ -- By definition of $\mathit{position}$ for $n > 0$
$\mathit{position}\ k\ (\mathit{replace}\ k\ v\ xs)$
$\equiv$ -- By inductive hypothesis
v

□

Theorem 5. *Given any state such that length state > 0, given $0 \leqslant i, j <$ length state where $i \not\equiv j$. For any $v, w \in \mathbb{V}$:*

$$\mathit{replace}\ i\ v\ (\mathit{replace}\ j\ w\ \mathit{state}) \equiv$$
$$\mathit{replace}\ j\ w\ (\mathit{replace}\ i\ v\ \mathit{state})$$

Proof. To prove that when replacing two values at different indices in the state, the order doesn't matter we use induction on i and j.

Base case: First we consider the simplest case where either $i = 0$ or $j = 0$. Without loss of generality, assume $i = 0$ and $j \not\equiv 0$:

$\mathit{replace}\ 0\ v\ (\mathit{replace}\ j\ w\ (x : xs))$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$
$\mathit{replace}\ 0\ v\ (x : \mathit{replace}\ (j-1)\ w\ xs)$
$\equiv$ -- By definition of $\mathit{replace}$ for $n = 0$
$v : \mathit{replace}\ (j-1)\ w\ xs$
$\equiv$
$\mathit{replace}\ j\ w\ (v : xs)$
$\equiv$
$\mathit{replace}\ j\ w\ (\mathit{replace}\ 0\ v\ (x : xs))$

Inductive step: For the induction step, assume the theorem holds for $i \leqslant k$ and $j \leqslant l$. We will show that the theorem holds for $i = k + 1$ and $j = l + 1$:

$\mathit{replace}\ (k+1)\ v\ (\mathit{replace}\ (l+1)\ w\ (x : xs))$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$
$\mathit{replace}\ (k+1)\ v\ (x : \mathit{replace}\ l\ w\ xs)$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$
$x : \mathit{replace}\ k\ v\ (\mathit{replace}\ l\ w\ xs)$
$\equiv$ -- By the induction hypothesis
$x : \mathit{replace}\ l\ w\ (\mathit{replace}\ k\ v\ xs)$
$\equiv$ -- Reverse using the definition of $\mathit{replace}$ for $n > 0$
$\mathit{replace}\ (l+1)\ w\ (x : \mathit{replace}\ k\ v\ xs)$
$\equiv$
$\mathit{replace}\ (l+1)\ w\ (\mathit{replace}\ (k+1)\ v\ (x : xs))$

□

Theorem 6. *Given any state such that length state > 0, given $0 \leqslant i, j <$ length state where $i \not\equiv j$. For any $v \in \mathbb{V}$:*

$$\mathit{position}\ j\ (\mathit{replace}\ i\ v\ \mathit{state}) \equiv \mathit{position}\ j\ \mathit{state}$$

Proof. To prove that retrieving the value at position j after replace the value at position i, where $i \not\equiv j$, we use induction on both i and j.

Base case: First we consider the simplest case where either $i = 0$ or $j = 0$. Without loss of generality, assume $j = 0$ and $i \not\equiv 0$:

$\mathit{position}\ 0\ (\mathit{replace}\ i\ v\ (x : \mathit{xs}))$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$
$\mathit{position}\ 0\ (x : \mathit{replace}\ (i-1)\ v\ \mathit{xs})$
$\equiv$ -- By definition of $\mathit{position}$ for $n = 0$
x
$\equiv$ -- By definition of $\mathit{position}$ for $n = 0$
$\mathit{position}\ 0\ (x : \mathit{xs})$

Inductive step: For the induction step, assume the theorem holds for $i \leqslant k$ and $j \leqslant l$. We will show that the theorem holds for $i = k+1$ and $j = l+1$:

$\mathit{position}\ (l+1)\ (\mathit{replace}\ (k+1)\ v\ (x : \mathit{xs}))$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$
$\mathit{position}\ (l+1)\ (x : \mathit{replace}\ k\ v\ \mathit{xs})$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$
$\mathit{position}\ l\ (\mathit{replace}\ k\ v\ \mathit{xs})$
$\equiv$ -- By the inductive hypothesis
$\mathit{position}\ l\ \mathit{xs}$
$\equiv$ -- By definition of $\mathit{replace}$ for $n > 0$ (reversed)
$\mathit{position}\ (l+1)\ (x : \mathit{xs})$

□

A.3 Global State

GS1

We want to prove that the following equivalence holds:

$\mathit{handle}_{\_}\ \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda v \rightarrow$
$\quad\quad \mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ v\ (\mathit{Pure}\ ())))))))$
$\quad \mathit{state}$
$\equiv$
$\mathit{handle}_{\_}\ (\mathit{Pure}\ ())\ \mathit{state}$

Proof.

$\mathit{handle}_{\_}\ \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda v \rightarrow$
$\quad\quad \mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ v\ (\mathit{Pure}\ ())))))))$
$\quad \mathit{state}$
$\equiv$ -- By definition of $\mathit{handle}_{\_}$ and by $\mathit{Corollary}$ 1.2
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a$
$\quad\quad (\mathit{unsafeCoerce}\ (\mathit{position}\ (l-i-1)\ \mathit{state}))$
$\quad\quad\quad (\mathit{Pure}\ ()))))$
$\quad \mathit{state}$
$\equiv$ -- By $\mathit{Corollary}$ 1.3
hFold
$\quad (\mathit{Pure}\ ())$
$\quad (\mathit{replace}\ (l-i-1)$
$\quad\quad (\mathit{unsafeCoerce}$
$\quad\quad\quad (\mathit{unsafeCoerce}\ (\mathit{position}\ (l-i-1)\ \mathit{state})))$
$\quad\quad \mathit{state})$
$\equiv$ -- Assuming $\mathit{unsafeCoerce}\ (\mathit{unsafeCoerce}\ x) = x$
-- By $\mathit{Theorem}$ 2:
-- $(\mathit{replace}\ i\ (\mathit{position}\ i\ \mathit{state})\ \mathit{state}) = \mathit{state}$
hFold
$\quad (\mathit{Pure}\ ())$
$\quad \mathit{state}$
$\equiv$ -- By definition of $\mathit{handle}_{\_}$
$\mathit{handle}_{\_}\ (\mathit{Pure}\ ())\ \mathit{state}$

□

GS2

We want to prove that the following equivalence holds:

$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda v \rightarrow$
$\quad\quad \mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \rightarrow \mathit{Pure}\ (v, w))))))))$
$\quad \mathit{state}$
$\equiv$
$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda v \rightarrow \mathit{Pure}\ (v, v)))))$
$\quad \mathit{state}$

Proof.

$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda v \rightarrow$
$\quad\quad \mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \rightarrow \mathit{Pure}\ (v, w))))))))$
$\quad \mathit{state}$
$\equiv$ -- By definition of $\mathit{handle}_{\_}$, and by $\mathit{Corollary}$ 1.2
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \rightarrow$
$\quad\quad \mathit{Pure}\ (\mathit{unsafeCoerce}$
$\quad\quad\quad (\mathit{position}\ (l-i-1)\ \mathit{state}), w)))))$
$\quad \mathit{state}$
$\equiv$ -- By $\mathit{Corollary}$ 1.2
hFold
$\quad (\mathit{Pure}\ (\mathit{unsafeCoerce}\ (\mathit{state}\ !!\ (l-i-1)),$
$\quad\quad \mathit{unsafeCoerce}\ (\mathit{position}\ (l-i-1)\ \mathit{state})))$
$\quad \mathit{state}$
$\equiv$ -- By $\mathit{Corollary}$ 1.2 (in reverse)
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda v \rightarrow \mathit{Pure}\ (v, v)))))$
$\quad \mathit{state}$
$\equiv$ -- By definition of $\mathit{handle}_{\_}$
$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda v \rightarrow \mathit{Pure}\ (v, v)))))$
$\quad \mathit{state}$

□

GS3

We want to prove that the following equivalence holds:

$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ v$
$\quad\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ w\ (\mathit{Pure}\ ())))))))$
$\quad \mathit{state}$
$\equiv$
$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ w\ (\mathit{Pure}\ ()))))$
$\quad \mathit{state}$

Proof.

$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ v$
$\quad\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ w\ (\mathit{Pure}\ ())))))))$

```
  state
≡   -- By definition of handle_ and by Corollary 1.3
hFold
  (Op (L (UpdateRef a w (Pure ()))))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡   -- By Corollary 1.3
hFold
  (Pure ())
  (replace (l − i − 1) (unsafeCoerce w)
    (replace (l − i − 1) (unsafeCoerce v) state))
≡
  -- By Theorem 3:
  -- replace i w (replace i v state) = replace i w state)
hFold
  (Pure ())
  (replace (l − i − 1) (unsafeCoerce w) (state))
≡   -- By Corollary 1.3 (reversed)
hFold
  (Op (L (UpdateRef a w (Pure ()))))
  state
≡   -- By definition of handle_
handle_
  hRefCoerced
  (Op (L (UpdateRef a w (Pure ()))))
  state
```

□

GS4
We want to prove that the following equivalence holds:

```
  handle_
    hRefCoerced
    (Op (L (UpdateRef a v
      (Op (L (DeRef a (λw → Pure w)))))))
    state
  ≡
  handle_
    hRefCoerced
    (Op (L (UpdateRef a v (Pure v))))
    state
```

Proof.

```
handle_
  hRefCoerced
  (Op (L (UpdateRef a v
        (Op (L (DeRef a (λw → Pure w)))))))
  state
≡   -- By definition of handle_, and by Corollary 1.3
hFold
  (Op (L (DeRef a (λw → Pure w))))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡   -- By Corollary 1.2
hFold
  (Pure
    (unsafeCoerce
      (position (l − i − 1)
        (replace (l − i − 1)
          (unsafeCoerce v) state))))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡
  -- By Theorem 4:
  -- position i (replace i x state) ≡ x
hFold
  (Pure (unsafeCoerce (unsafeCoerce v)))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡
  -- Assuming unsafeCoerce (unsafeCoerce x) ≡ x
hFold
  (Pure v)
  (replace (l − i − 1) (unsafeCoerce v) state)
≡   -- By Corollary 1.3 (reversed)
hFold
  (Op (L (UpdateRef a v (Pure v))))
  state
≡   -- By definition of handle_
handle_
  hRefCoerced
  (Op (L (UpdateRef a v (Pure v))))
  state
```

□

GS5
We want to prove that the following equivalence holds:

```
  handle_
    hRefCoerced
    (Op (L (DeRef a (λv →
      Op (L (DeRef b (λw → Pure (v, w))))))))
    state
  ≡
  handle_
    hRefCoerced
    Op (L (DeRef b (λw →
      Op (L (DeRef a (λv → Pure (v, w)))))))
    state
```

Proof.

```
handle_
  hRefCoerced
  (Op (L (DeRef a (λv →
    Op (L (DeRef b (λw → Pure (v, w))))))))
  state
≡   -- By definition of handle, and by Corollary 1.2
hFold
  (Op (L (DeRef b (λw →
    Pure
      (unsafeCoerce
        (position (l − i − 1) state), w)))))
  state
≡   -- By Corollary 1.2
hFold
  (Pure
    (unsafeCoerce (position (l − i − 1) state),
      unsafeCoerce (position (l − j − 1) state)))
  state
≡   -- By Corollary 1.2 (reversed)
hFold
  (Op (L (DeRef a (λv →
    (v, unsafeCoerce (position (l − j − 1) state))))))
  state
≡   -- By Corollary 1.2 (reversed)
hFold
  Op (L (DeRef b (λw →
    Op (L (DeRef a (λv → Pure (v, w)))))))
  state
≡   -- By definition of handle
handle_
  hRefCoerced
  Op (L (DeRef b (λw →
    Op (L (DeRef a (λv → Pure (v, w)))))))
  state
```

□

GS6
We want to prove that the following equivalence holds:

```
  handle_
    hRefCoerced
    (Op (L (UpdateRef a v
      (Op (L (UpdateRef b w (Pure ()))))))))
    state
  ≡
  handle_
    hRefCoerced
    (Op (L (UpdateRef b w
      (Op (L (UpdateRef a v (Pure ()))))))))
    state
```

Proof.

```
handle_
  hRefCoerced
  (Op (L (UpdateRef a v
    (Op (L (UpdateRef b w (Pure ()))))))))
  state
≡   -- By definition of handle, and by Corollary 1.3
hFold
  (Op (L (UpdateRef b w (Pure ()))))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡   -- By Corollary 1.3
hFold
  (Pure ())
  (replace (l − j − 1) (unsafeCoerce w)
    (replace (l − i − 1) (unsafeCoerce v) state))
≡
  -- By Theorem 5:
  --     replace i v (replace j w state)
  --       ≡ replace j w (replace i v state)
  -- This assumes i ≢ j, otherwise this would not hold.
hFold
  (Pure ())
  (replace (l − i − 1) (unsafeCoerce v)
    (replace (l − j − 1) (unsafeCoerce w) state))
≡   -- By Corollary 1.3 (reversed)
hFold
  (Op (L (UpdateRef a v (Pure ()))))
  (replace (l − j − 1) (unsafeCoerce w) state)
≡   -- By Corollary 1.3 (reversed)
hFold
  (Op (L (UpdateRef b w
    (Op (L (UpdateRef a v (Pure ()))))))))
  state
≡   -- By definition of handle
handle_
  hRefCoerced
  (Op (L (UpdateRef b w
    (Op (L (UpdateRef a v (Pure ()))))))))
  state
```

□

GS7
We want to prove that the following equivalence holds:

```
  handle_
    hRefCoerced
    (Op (L (UpdateRef a v
      (Op (L (DeRef b (λw → Pure w)))))))
    state
  ≡
  handle_
    hRefCoerced
    (Op (L (DeRef b (λw →
      Op (L (UpdateRef a v (Pure w)))))))
    state
```

Proof.

```
handle_
  hRefCoerced
  (Op (L (UpdateRef a v
    (Op (L (DeRef b (λw → Pure w)))))))
  state
≡   -- By definition of handle, and by Corollary 1.3
hFold
  (Op (L (DeRef b (λw → Pure w))))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡   -- By Corollary 1.2
hFold
  Pure
    (unsafeCoerce (position (l − j − 1)
      (replace (l − i − 1) (unsafeCoerce v) state)))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡
  -- By Theorem 6:
  --     position j (replace i x state)
  --       ≡ position j state
  -- This assumes i ≢ j or this won't hold.
hFold
  Pure (unsafeCoerce (position (l − j − 1) state))
  (replace (l − i − 1) (unsafeCoerce v) state)
≡   -- By Corollary 1.3
hFold
  Op (L (UpdateRef a v (Pure
    (unsafeCoerce (position (l − j − 1) state)))))
  state
≡   -- By Corollary 1.2
hFold
  Op (L (DeRef b (λw →
    Op (L (UpdateRef a v (Pure w))))))
  state
≡   -- By definition of handle_
handle_
  hRefCoerced
  (Op (L (DeRef b (λw →
    Op (L (UpdateRef a v (Pure w))))))
  state
```

□

A.4 Theory of Block

B1

We want to prove that the following equivalence holds:

$$\begin{array}{l}
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad \mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to \mathit{Pure}\ ())))\\
\quad \mathit{state}\\
\equiv\\
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Pure}\ ())\\
\quad (\mathit{unsafeCoerce}\ v : \mathit{state})
\end{array}$$

Proof.

$$\begin{array}{l}
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad \mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to \mathit{Pure}\ ())))\\
\quad \mathit{state}\\
\equiv \quad \text{-- By definition of } \mathit{handle}\text{, and by } \mathit{Corollary}\ 1.1\\
\mathit{hFold}\\
\quad (\mathit{Pure}\ ())\\
\quad (\mathit{unsafeCoerce}\ v : \mathit{state})\\
\equiv \quad \text{-- By definition of } \mathit{handle}\\
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Pure}\ ())\\
\quad (\mathit{unsafeCoerce}\ v : \mathit{state})
\end{array}$$

□

B2

We want to prove that the following equivalence holds:

$$\begin{array}{l}
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to\\
\qquad \mathit{Op}\ (L\ (\mathit{MkRef}\ w\ (\lambda b \to \mathit{Pure}\ (a, b))))))))\\
\quad \mathit{state}\\
\equiv\\
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ w\ (\lambda b \to\\
\qquad \mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to \mathit{Pure}\ (a, b))))))))\\
\quad \mathit{state}
\end{array}$$

Proof. We start by unfolding the first expression of the equivalence to arrive at the following:

$$\begin{array}{l}
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to\\
\qquad \mathit{Op}\ (L\ (\mathit{MkRef}\ w\ (\lambda b \to \mathit{Pure}\ (a, b))))))))\\
\quad \mathit{state}\\
\equiv\\
\quad \text{-- By definition of } \mathit{handle}\text{, and by } \mathit{Corollary}\ 1.1\\
\mathit{hFold}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ w\ (\lambda b \to \mathit{Pure}\ (\mathit{IntRef}\ l, b)))))\\
\quad (\mathit{unsafeCoerce}\ v : \mathit{state})\\
\equiv \quad \text{-- By } \mathit{Corollary}\ 1.1\\
\mathit{hFold}\\
\quad (\mathit{Pure}\ (\mathit{IntRef}\ l, \mathit{IntRef}\ (l + 1)))\\
\quad (\mathit{unsafeCoerce}\ w : \mathit{unsafeCoerce}\ v : \mathit{state})
\end{array}$$

Next we look at the second expression of the equivalence and arrive at the following:

$$\begin{array}{l}
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ w\ (\lambda b \to\\
\qquad \mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to \mathit{Pure}\ (a, b))))))))\\
\quad \mathit{state}\\
\equiv\\
\quad \text{-- By definition of } \mathit{handle}\text{, and by } \mathit{Corollary}\ 1.1\\
\mathit{hFold}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to \mathit{Pure}\ (a, \mathit{IntRef}\ l)))))\\
\quad (\mathit{unsafeCoerce}\ w : \mathit{state})\\
\equiv \quad \text{-- By } \mathit{Corollary}\ 1.1\\
\mathit{hFold}\\
\quad (\mathit{Pure}\ (\mathit{IntRef}\ (l + 1), \mathit{IntRef}\ l))\\
\quad (\mathit{unsafeCoerce}\ v : \mathit{unsafeCoerce}\ w : \mathit{state})
\end{array}$$

Although the expressions $\mathit{Pure}\ (\mathit{IntRef}\ l, \mathit{IntRef}\ (l+1))$ and $\mathit{Pure}\ (\mathit{IntRef}\ (l+1), \mathit{IntRef}\ l)$ look different at first glance, using these references in either case would result in the same outcome. This is because references a and b are essentially swapped, but their relative positions and subsequent usage remain consistent. The Replace & Position theorems support this by showing that the order of independent operations (like creating references) does not affect the final state.

□

B3$_\mathbf{n}$

We want to prove that the following block state law holds:

$$\begin{array}{l}
v : \mathbb{V}, \vec{a} : \mathbb{A}^{\otimes n} \vdash \mathbf{do}\ \vec{b} \leftarrow \mathit{mkref}^{n}(\vec{a}, v)\mathbf{;}\ \mathit{return}\ p_n(\vec{b})\\
\quad \equiv \mathbf{do}\ b \leftarrow \mathit{mkref}\ v\mathbf{;}\ \mathit{return}\ (\vec{a}, b) : \mathbb{A}^{\otimes n} \times \mathbb{A}
\end{array}$$

As previously discussed, the mkref^n function is defined to allocate a new location different from all other references passed as its input. For our handler implementation, this is always the case, as it utilizes the entire state to allocate a new location, ensuring the uniqueness of each allocated location. Consequently, this law is trivially true for our implementation. We argue that this law is trivially true based on the fact that each reference created using our handler will be a unique reference into the store.

A.5 Local State

LS1

We want to prove that the following equivalence holds:

$$\begin{array}{l}
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to\\
\qquad \mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ w\ (\mathit{Pure}\ a)))))))\\
\quad \mathit{state}\\
\equiv\\
\mathit{handle}_{\_}\ \mathit{hRefCoerced}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ w\ (\lambda a \to \mathit{Pure}\ a)))
\end{array}$$

Proof.

$$\begin{array}{l}
\mathit{handle}_{\_}\\
\quad \mathit{hRefCoerced}\\
\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda a \to\\
\qquad \mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ w\ (\mathit{Pure}\ a)))))))\\
\quad \mathit{state}
\end{array}$$

```
 ≡
   -- By definition of handle_, and by Corollary 1.1
hFold
  (Op (L (UpdateRef (IntRef l) w
     (Pure (IntRef l)))))
  (unsafeCoerce v : state)
 ≡
(hFold (Pure (IntRef l)))
  (replace ((l + 1) − l − 1) (unsafeCoerce w)
     (unsafeCoerce v : state))
 ≡   -- By definition of hFold and fold
(ret_ hRefCoerced)
  (IntRef l)
  (replace 0 (unsafeCoerce w)
     (unsafeCoerce v : state))
 ≡   -- By definition of replace
(ret_ hRefCoerced)
  (IntRef l)
  (unsafeCoerce w : state)
 ≡   -- By definition of hFold and fold (reversed)
(hFold (Pure (IntRef l)))
  (unsafeCoerce w : state)
 ≡   -- By Corollary 1.1 (reversed)
hFold
  (Op (L (MkRef w (λa → Pure a))))
  state
 ≡   -- By definition of handle_
handle_
  hRefCoerced
  (Op (L (MkRef v (λa →
     Op (L (UpdateRef a w (Pure a)))))))
  state
```

□

LS2
We want to prove that the following equivalence holds:

```
handle_
  hRefCoerced
  (Op (L (MkRef v (λa →
     Op (L (DeRef a (λw → Pure (w, a))))))))
  state
 ≡
handle_
  hRefCoerced
  (Op (L (MkRef v (λa → Pure (v, a)))))
  state
```

Proof.

```
handle_
  hRefCoerced
  (Op (L (MkRef v (λa →
     Op (L (DeRef a (λw → Pure (w, a))))))))
  state
 ≡
  -- By definition of handle_ and by Corollary 1.1
hFold
  (Op (L (DeRef (IntRef l) (λw →
     Pure (w, IntRef l)))))
  (unsafeCoerce v : state)
 ≡   -- By Corollary 1.2
hFold
  (Pure (unsafeCoerce (position ((l + 1) − l − 1)
     (unsafeCoerce v : state)), IntRef l))
  (unsafeCoerce v : state)
 ≡
hFold
  (Pure (unsafeCoerce (position 0
     (unsafeCoerce v : state)), IntRef l))
  (unsafeCoerce v : state)
 ≡
hFold
  (Pure (unsafeCoerce
     (unsafeCoerce v), IntRef l))
  (unsafeCoerce v : state)
 ≡
  -- Assuming unsafeCoerce (unsafeCoerce x) ≡ x
hFold
  (Pure (v, IntRef l))
  (unsafeCoerce v : state)
 ≡   -- By Corollary 1.1 (reversed)
hFold
  (Op (L (MkRef v (λa → Pure (v, a)))))
  state
 ≡   -- By definition of handle_
handle_
  hRefCoerced
  (Op (L (MkRef v (λa → Pure (v, a)))))
  state
```

□

LS3
We want to prove that the following equivalence holds:

```
handle_
  hRefCoerced
  (Op (L (MkRef v (λb →
     Op (L (UpdateRef a w (Pure b)))))))
  state
 ≡
handle_
  hRefCoerced
  (Op (L (UpdateRef a w
     (Op (L (MkRef v (λb → Pure b)))))))
  state
```

Proof.

```
handle_
  hRefCoerced
  (Op (L (MkRef v (λb →
     Op (L (UpdateRef a w (Pure b)))))))
  state
 ≡   -- By definition of handle_ and Corollary 1.1
hFold
     (Op (L (UpdateRef a w (Pure (IntRef l)))))
     (unsafeCoerce v : state)
 ≡
  -- Assuming a = IntRef i, and i < l
hFold
  (Pure (IntRef l))
  (replace ((l + 1) − i − 1) (unsafeCoerce w)
     (unsafeCoerce v : state))
 ≡
  -- We know i < l, since we just increased the size by 1
```

-- Since we know that $i < l$, we know that
-- $((l+1) - i - 1) > 0$. By the definition of $\mathit{replace}$,
-- we know that for the case where $n > 0$:
-- $\quad \mathit{replace}\ (k+1)\ z\ (x : \mathit{xs})$
-- $\quad\quad \equiv x : \mathit{replace}\ k\ \mathit{xs}$
hFold
$\quad (\mathit{Pure}\ (\mathit{IntRef}\ l))$
$\quad (\mathit{unsafeCoerce}\ v : (\mathit{replace}\ (l - i - 1)$
$\quad\quad (\mathit{unsafeCoerce}\ w)\ \mathit{state}))$
$\equiv$ -- By $\mathit{Corollary}$ 1.1 (reversed)
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to \mathit{Pure}\ b))))$
$\quad (\mathit{replace}\ (l - i - 1)$
$\quad\quad (\mathit{unsafeCoerce}\ w)\ \mathit{state})$
$\equiv$ -- By $\mathit{Corollary}$ 1.3
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ w$
$\quad\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to \mathit{Pure}\ b)))))))$
$\quad \mathit{state}$
$\equiv$ -- By definition of $\mathit{handle}_{\_}$
$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{UpdateRef}\ a\ w$
$\quad\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to \mathit{Pure}\ b)))))))$
$\quad \mathit{state}$

□

LS4

We want to prove that the following equivalence holds:

$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to$
$\quad\quad \mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \to \mathit{Pure}\ (w, b)))))))))$
$\quad \mathit{state}$
$\equiv$
$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \to$
$\quad\quad \mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to \mathit{Pure}\ (w, b)))))))))$
$\quad \mathit{state}$

Proof.

$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to$
$\quad\quad \mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \to \mathit{Pure}\ (w, b)))))))))$
$\quad \mathit{state}$
$\equiv$
-- By definition of $\mathit{handle}_{\_}$, and by $\mathit{Corollary}$ 1.1
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \to \mathit{Pure}\ (w, \mathit{IntRef}\ l)))))$
$\quad (\mathit{unsafeCoerce}\ v : \mathit{state})$
$\equiv$
-- By $\mathit{Corollary}$ 1.2, assuming $a = \mathit{IntRef}\ i$, $i < l$
hFold
$\quad (\mathit{Pure}\ (\mathit{unsafeCoerce}\ (\mathit{position}\ ((l+1) - i - 1)$
$\quad\quad (\mathit{unsafeCoerce}\ v : \mathit{state})), \mathit{IntRef}\ l))$
$\quad (\mathit{unsafeCoerce}\ v : \mathit{state})$
$\equiv$
-- We know $i < l$, since we just increased the size by 1
-- By the definition of $\mathit{replace}$, we know that for the
-- case where $n > 0$:
-- $\quad \mathit{replace}\ (k+1)\ z\ (x : \mathit{xs})$
-- $\quad\quad \equiv x : \mathit{replace}\ k\ \mathit{xs}$
hFold
$\quad (\mathit{Pure}\ (\mathit{unsafeCoerce}$
$\quad\quad (\mathit{position}\ (l - i - 1)\ \mathit{state}), \mathit{IntRef}\ l))$
$\quad (\mathit{unsafeCoerce}\ v : \mathit{state})$
$\equiv$ -- By $\mathit{Corollary}$ 1.1 (reversed)
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to$
$\quad\quad (\mathit{Pure}\ (\mathit{position}\ (l - i - 1)\ \mathit{state}), b)))))$
$\quad \mathit{state}$
$\equiv$ -- By $\mathit{Corollary}$ 1.2 (reversed)
hFold
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \to$
$\quad\quad \mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to \mathit{Pure}\ (w, b)))))))))$
$\quad \mathit{state}$
$\equiv$ -- By definition of $\mathit{handle}_{\_}$
$\mathit{handle}_{\_}$
$\quad \mathit{hRefCoerced}$
$\quad (\mathit{Op}\ (L\ (\mathit{DeRef}\ a\ (\lambda w \to$
$\quad\quad \mathit{Op}\ (L\ (\mathit{MkRef}\ v\ (\lambda b \to \mathit{Pure}\ (w, b)))))))))$
$\quad \mathit{state}$

□