An Algebraic Effect for ML-Style References in Haskell

Author – Daan Panis **Supervisors** – Casper Bach Poulsen, Jaro Reinders **Examiner** – Annibale Panichella

(1) Introduction & Background

Mistakes in software can be very expensive. A common cause of mistakes are side-effecting operations, like exceptions and I/O.

Traditionally, Haskell handles side effects using monads and monad transformers, which can obscure how effects are handled and complicate reasoning about programs.

Algebraic effects **[1]** offer an alternative by separating the syntax and semantics of side-effecting operations from how the operations are handled.

The free monad represents computations as abstract syntax trees (ASTs), allowing the creation of algebraic effects through functor data types.

Handlers define how specific operations are processed while keeping other operations in the AST intact, enabling a modular approach to managing side effects with customisable handlers.

A notable side effect is mutable references, similar to those in ML **[2]**. They enable you to allocate a new value in a store, returning a pointer to that location, retrieve a value given a pointer, and update the value at that location. Staton proposed laws for such mutable operations to ensure predictable and correct behaviour **[3]**.

(2) Goal of Research

The goal of this research is to answer the question:

> How can ML-style references **[2]** be implemented in Haskell using algebraic effects that adhere to global, block, and local state laws as defined by Staton **[3]**?

By answering the question we offer programmers a more flexible and predictable way of using mutable references in Haskell.

It enables you to write programs and data structures that are normally hard to write in Haskell. Staton's state laws **[3]** give you a framework to reason about the programs you write, ensuring their correctness. It offers easy integration into larger systems, such as domain-specific languages (DSLs), adding useful effects while allowing reasoning about the programs.

(3) Implementation

Our implementation, built on top of Bach Poulsen's blog post on free monads and algebraic effects **[4]**, defines three operations: creating, updating, and reading a reference. We also introduce smart constructors for ease of use.

```
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

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 ())))
```

The handler for this effect uses a simple list as a store and uses a wrapper around an integer to point to a location in the list.

(4) Staton's State Laws

Staton describes laws that govern mutable references to ensure that operations on mutable state in programming languages behave predictably and correctly **[3]**. We slightly modified these laws to align with Haskell's syntax and our mutable reference operations.

```
GS1. do v <- deref a; update a v ≡ return ()
GS2. do v <- deref a; w <- deref a; return (v, w)
      ≡ do v <- deref a; return (v, v)
GS3. do update a v; update a w; ≡ do update a w
GS4. do update a v; w <- deref a; do update a v; return w
      ≡ do update a v; return v
GS5. do v <- deref a; w <- deref b; return (v, w)
      ≡ do w <- deref b; v <- deref a; return (v, w)
GS6. do update a v; update b w; ≡ update b w; update a v;

B1. do a  <- mkref v; return () ≡ return ()
B2. do a  <- mkref v; b <- mkref w; return (a, b)
      ≡ do b <- mkref w; a <- mkref v; return (a, b)
B3. do bs <- mkref_n(as, v); return p_n(bs)
      ≡ b <- mkref v; return (as, b)

LS1. do a <- mkref v; update a w; return a
      ≡ do a <- mkref w; return a
LS2. do a <- mkref v; w <- deref a; return (w, a)
      ≡ do a <- mkref v; return (v, a)
LS3. do b <- mkref v; update a w; return b
      ≡ do update a w; b <- mkref v; return b
LS4. do b <- mkref v; w <- deref a; return (w, b)
      ≡ do w <- deref a; b <- mkref v; return (w, b)
```

(5) Proving Programs

With the operations and smart constructors for mutable references, we can no write programs in an imperative style like the program that computes the factorial of a number n.

```
impFact :: Int -> Free (MLRef ref + End) Int
impFact n = do
  x <- mkref 1; acc <- mkref 1
  while
    (deref x >= return . (<= n))
    (do x' <- deref x; acc <- deref acc
        update acc (acc' * x'); update x (x' + 1))

  deref acc
```

Using Staton's state laws, we can reorder and transform operations to derive the definition of factorial, thereby proving the correctness of the program.

(6) Discussion & Conclusion

This research successfully implemented an algebraic effect for ML-style references, proving the handler adheres to Staton's state laws and demonstrating program verification.

The main limitation is performance. The chosen basic handler, while easy to prove, is slow.

We recommend further research into faster handlers using efficient data structures or alternatives like IORef, and studying interactions with other effects, such as exceptions.

Additionally, exploring optimisations using Staton's state laws could enhance competitiveness.

(7) References

[1] Plotkin, G., Power, J. (2001). Adequacy for Algebraic Effects. In: Honsell, F., Miculan, M. (eds) Foundations of Software Science and Computation Structures. FoSSaCS 2001. Lecture Notes in Computer Science, vol 2030. Springer, Berlin, Heidelberg.

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

[3] Staton, S. (2010). Completeness for Algebraic Theories of Local State. In: Ong, L. (eds) Foundations of Software Science and Computational Structures. FoSSaCS 2010. Lecture Notes in Computer Science, vol 6014. Springer, Berlin, Heidelberg.

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