Algebraic Effects and Handlers for Software Transactional Memory

Matej Tomášek
Supervisor(s): Casper Bach Poulsen, Jaro Reinders
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: Matej Tomášek
Final project course: CSE3000 Research Project
Thesis committee: Casper Bach Poulsen, Jaro Reinders, Annibale Panichella

Abstract

Algebraic effects and handlers has been a popular approach for modelling side-effects in functional programming languages. Focusing on composability and modularity, this approach separates the effectful syntax from its semantics, which helps programmers to create effect abstractions such that their implementation can be modified without changing the syntax. However, there exist mainstream functional programming languages, like Haskell, which lack built-in frameworks to accommodate specifying side-effects in this manner. In this paper, we provide an interface for Haskell's *Software Transactional Memory* (STM), a concurrency abstraction, in the framework of algebraic effects and handlers from prior literature. We embed our implementation into a simple concurrency model using higher-order effects, in order to demonstrate it is possible to define and execute effectful concurrent programs that obey the semantics of Haskell's STM. Furthermore, we prove that our implementation satisfies the necessary laws governing our interface, such that programmers can easily reason about programs using our STM model.

1 Introduction

In purely functional programming, *monads* [7] have been the standard approach for modelling and building programs with side-effects. These include I/O operations, exceptions, mutable state, non-determinism, concurrency and more. More recently, however, an increasingly attractive technique of modelling computational effects has emerged in the form of *algebraic effects* [10] and their *handlers* [11]. Using this approach, one can separately define the effectful operations (syntax) following algebraic equations from the handlers implementing the algebra, which gives way to concisely combining multiple effects in a single program.

For example, by declaring `throw` to be an operation that raises an exception with a given message, and `get` and `put` to be operations for getting and setting the contents of a mutable state holding an `Int`, we can define a program for safely dividing the number in the mutable state shown in Figure 1 (omitting the function signature). In order to execute the program, we have to implement handlers for each effect. Here, the handler `hExc` defines the effects of the `throw` operation, and the handler `hState` s defines the effects of the `get` and `put` operations on a state s. In Figure 2, we evaluate our program by applying these handlers in sequence, which outputs either the return value paired value currently held by the state, or the message from a thrown exception (depending on the actual implementation).

```
safeDiv d = do
    n <- get
    if d == 0 then
        throw "Division by 0!"
    else do
        set (n `div` d)
        get
```

Figure 1: The program for safe division

```
> hExc (hState 4 (safeDiv 2))
Right (2, 2)

> hExc (hState 4 (safeDiv 0))
Left "Division by 0!"
```

Figure 2: Evaluations of the program

Such out-of-the-box modularity is not available with the traditional monadic approach.

Additionally, a different order of handler applications may in turn imply a different interpretation of the underlying program. This is the case for our handlers `hState` and

hExc. If we were to extend our example by declaring a *scoped* operation catch m n that catches potential exceptions in m and continues with n (handled by hExc), and consider a new program called transact [19] from Figure 3, then swapping the application order of handler would yield to different results as shown in Figure 4.

```
transact = do
    put 1
    catch
        (put 2 >> throw "Error")
        get
```

Figure 3: The transactional program

```
> hExc (hState 0 transact)
Right (1, 1)

> hState 0 (hExc transact)
(Right 2, 2)
```

Figure 4: Different orders of evaluation

Thus, for the first ordering, we can observe so-called transactional semantics, discarding the changes to the state inside the catch, between the these two effects by virtue of catch being scoped, i.e. containing inner computations [20].

Although there has been extensive research of algebraic effects and handlers in recent years [10] [3], there are functional programming languages, like Haskell, which lack meaningful built-in implementations for specifying computational effects this way. Fortunately, Haskell's community built multiple libraries [1] [2] for implementing algebraic effects, with the literature also offering proposing frameworks to model algebraic effects [12], and even higher-order effects [2] [20]. These frameworks are mainly based on the *free monad* approach [15].

One of the more famous transactional effects in Haskell is Software Transactional Memory (STM) [8]. It is a concurrency abstraction where threads can interact through memory, using database-like transactions instead of locks. Its main feature is that it allows to easily compose concurrent program fragments into a larger fragment without the programmer worrying about possible failures, e.g. deadlock. We would like to explore how STM would look like in the framework of algebraic effects and handlers, to what extent it resembles transactional interactions of different effects, and how other effects can be extended with similar semantics as those of the STM. Thus, the purpose of this research paper is to answer the following question: *"How can we implement and reason about an algebraic effect model of STM in Haskell?"*, combining the previously outlined concepts. More precisely, the following sub-questions have been identified:

1. How can an implementation of algebraic effects that implements the intended behavior of STM look like in correspondence with the literature?
2. What are the mathematical laws describing the intended behavior of STM and prove the proposed implementation is correct with respect to them?
3. How do the operations of our STM interface interact with the operations of other effects, and what the "extension" of transactional memory to other effects looks like?

There are several contributions made in this paper which aid us in answering these research questions:

- We introduce a framework for defining algebraic effects and handlers in Haskell formulated in [12].

[1] https://hackage.haskell.org/package/fused-effects
[2] https://hackage.haskell.org/package/freer-simple

- In this framework, we propose an abstraction of STM, modelling its semantics according to [5], and demonstrating it on an example program in a concurrent setting.
- We establish a technique of verifying our implementation in a formal way, i.e. proving that it respects the algebraic theory of STM operations from [4].
- We analyze the composition of the STM effect with different effects, how changing the order of handling changes the semantics.

The rest of this paper is structured as follows. We start with a problem description regarding the framework for algebraic effects and handlers in Section 2, as well as Haskell's STM abstraction and its operational semantics. In Section 3, we outline the implementation of our abstraction of STM as an algebraic effect. We verify our implementation against a formal reasoning technique for STM abstractions in Section 4, and in Section 5, we apply the implementation to an example program and propose the extension of transactional memory to other effects. In Section 6, we talk about ethical considerations and reproducibility of our research. The discussion of our implementation in a broader context is provided in Section 7. Section 8 describes related work. And finally, the conclusion and the overview of possible future improvements are in Section 9.

2 Background and Methodology

This section offers a complete overview of the problem tackled in this research paper, namely the implementation of algebraic effects and handlers for Software Transactional Memory in Haskell. Choosing a problem-oriented approach, we focus on presenting the necessary background concepts that form the basis of our research. The aim is to provide the necessary context to the methodology with which we will develop our implementation. We start with providing a short introduction to algebraic effects and handlers and their implementation in Haskell. Afterwards, we explain the operational semantics of STM Haskell and how it differs from lock-based concurrency.

2.1 Algebraic Effects and Handlers in Haskell

Unfortunately, Haskell does not provide a way to implement algebraic effects out of the box. In order to do so, we adopt an "à la carte" approach of expressing effectful programs as free monads borrowed from [12] that was inspired by [15].

`Free` monads represent abstract syntax trees, where `Pure` corresponds to a value, and `Op` corresponds to an effectful operation given by a *signature functor* `f`. In general, the type `f k` encodes a the syntax of an operation whose continuation is of type `k`, so `f (Free f a)` represents an operation whose continuation is the syntax tree itself `Free f a`. As was previously mentioned, these programs may be written against more than one syntax interface. They are modelled as rows of effect types in `Free` using the functor co-product `+`. Additionally, we intend for the programmer to easily compose individual operations together. Thus, we introduce *signature subtyping* with the type class `<` to automatically inject a piece of syntax into a larger row of effects, which allows us to define *smart constructors* for any effect operation. Finally, free monads provide semantics to the encoded effects by (recursively) *folding* over its syntax tree. The function `fold` uses a generator function `(a -> b)` to transform values from `Pure a` to `b`, and an algebra `(f b -> b)` to extract recursively-folded continuations from the signature functor of type `b`. Figure 5 encapsulates this entire framework.

```haskell
data Free f a where
    Pure :: a -> Free f a
    Op   :: f (Free f a) -> Free f a

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

```haskell
class f < g where
    inj :: f k -> g k

fold :: Functor f
    => (a -> b) -> (f b -> b)
    -> Free f a -> b
```

Figure 5: Free monad framework

`fold` is also used to implement the instance of `Monad`, `Applicative` and `Functor` for `Free`. To mark the end of the effect row, we use the `End` functor with no constructors and the function `un :: Free End a -> a` to extract the result `a`, after handling all other effects before `End`. For the sake of brevity, we do not include their definitions in this paper.

For example, in Figure 6, the syntax of `State` effect can be represented by a signature functor and smart constructors from [12].

```haskell
data State s k where
    Get :: (s -> k) -> State s k
    Put :: s -> k -> State s k
    deriving Functor
```

```haskell
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 ())
```

Figure 6: State effect signature

An effect handler implements a specific interface by defining the interpretation of each operation associated with an effect. The idea is that a handler should forward the rest of the computation to other handlers, thus we can model handlers as the following:

```haskell
data Handler f a f' b = Handler {
    ret  :: a -> Free f' b,
    hdlr :: f (Free f' b) -> Free f' b
}
```

```haskell
handle :: (Functor f, Functor f')
    => Handler f a f' b
    -> Free (f + f') a
    -> Free f' b
```

Figure 7: The handler framework

The function `handle` takes a handler `Handler f a f' b`, that handles an effect `f`, leaving behind the effects `f'`, and transforming the return type of the computation from `a` to `b`, by the means of `fold`.

For the sake of brevity, the definitions of `StatefulHandler` with its handling function and `hState` are not included here as well. All omitted code of our paper is available in [16] and in [12].

2.2 Software Transactional Memory in Haskell

Software Transactional Memory is a programming abstraction for shared-memory concurrency [5]. Most prevalent form of concurrency is *lock-based concurrency*, although there are several fundamental drawbacks to this approach

- As simple as the approach might be, lock-based programs do not scale well. The problem of ensuring correctness, liveness and good performance becomes unmanagable with complexity

- Lock-based programs do not compose. Even though, the fragments of the concurrent program are correct, they may fail when combined in isolation, thus we have to expose the locking protocol to merge them correctly

Basing concurrency control on atomic memory transactions alleviates these problems, as the abstractions are separated from the semantics of concurrency. The key insight is that a block of code, including nested calls, enclosed in `atomically`, is guaranteed to run atomically with respect to every other block. More precisely, it makes the following guarantees:

1. Atomicity — the effects of one block become visible to other threads all at once
2. Isolation — the block is completely unaffected by other threads

The transactional memory shared between the atomic blocks is traditionally implemented using *optimistic synchronisation*. Each block takes snapshot of the memory at the beginning of its execution and then executes against that snapshot, keeping track of all reads and writes in a thread-local log. Once the block finishes execution, it validates that the initial view of the memory is consistent with its current contents - i.e. no concurrent transaction has committed conflicting changes. If so, it commits the changes recorded in the log to memory, otherwise, the transaction aborts and runs at a later time.

In Figure 8, we outline operations of Haskell's STM interface [8] (not including `throw` and `catch` operations from [5]).

```
atomically :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a

newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Figure 8: The STM interface

We can interact with the transactional memory using *transactional variables* (`TVar`). The `newTVar`, `readTVar` and `writeTVar` return `STM` actions, since only memory actions and pure computations can be performed in atomic blocks - this is guaranteed by the virtue of `STM` being a *monad*. Additionally, `STM` action cannot be executed without the protection of `atomically`.

Operations `retry` and `orElse` represent concurrency primitives *blocking* and *choice*. Conceptually, `retry` aborts the transaction with no effect and restarts it at the beginning. `orElse` lets us compose transactions as alternatives - i.e. if the first one retries, it is abandoned with no effect and the second transaction is run instead.

The transactional semantics of STM also simplify the process of formal reasoning about concurrency, as opposed to lock-based approach [4], which we utilize in Section 4.

3 Implementing the STM Interface

This section outline the main contribution of this paper, as we present the implementation of STM in the framework of algebraic effects and handlers from 2.1.

3.1 Effect Interface and Single-Threaded Handler

In order to define our effect interface, we need to model the structure and behaviour of the transactional memory. In Haskell's STM, this is done using optimistic synchronisation,

which requires validation and commiting of thread-local transaction logs. In the single-threaded case, there is no need for validating a transaction log, or keeping a log for that matter. Since, there are no transactions running in parallel that can commit their changes before validation happens, it is evident all transactions on a single thread are executed in sequence. Therefore, the reads and write of single-threaded transactions are immediately reflected in the transactional memory, which is in correspondence with the formal reasoning [4]. In this section, we initially focus on implementing the effect on single-threaded programs, i.e. without `atomically`, as it provides the necessary foundation that can be easily extended into multi-threaded environment with optimistic synchronisation.

Since `newTVar` can allocate `TVar`s of any type into the transactional memory, our model must be able to store values of any type in easily accessible and orderly manner. The simplest solution is to represent the transactional memory as a list of dynamic cells, which we name `Heap` [4] (Figure 9), and the transactional variables become references to these cells. Also, `Heap` should be an instance of `Eq`, as later on we will require means of comparing snapshots of the transactional memory to properly implement optimistic synchronisation. In order to perform type-safe reads and writes to any arbitrary cell in the heap, we define helper functions `alloc`, `update` and `lookup`. Thus, we can now introduce the signature functor of our `STM` effect (and its smart constructors) that corresponds very closely to Haskell's STM interface (Figure 11).

```
data Cell = forall a. (Show a, Eq a)
    => Cell a

deriving instance Show Cell

instance Eq Cell where
    (==) :: Cell -> Cell -> Bool
    Cell a == Cell b =
        show a == show b

newtype TVar a = TVar Int
    deriving (Eq, Show)

type Heap = [Cell]
```

```
alloc :: (Show a, Eq a)
    => a -> Heap -> (TVar a, Heap)

update :: (Show a, Eq a)
    => TVar a -> a -> Heap -> Heap

lookup :: (Show a, Eq a)
    => TVar a -> Heap -> a
```

Figure 9: Our `Heap` implementation

```
data STM k where
    New     :: (Show a, Eq a) => a -> (TVar a -> k) -> STM k
    Read    :: (Show a, Eq a) => TVar a -> (a -> k) -> STM k
    Write   :: (Show a, Eq a) => TVar a -> a -> k -> STM k
    Retry   :: STM k
    OrElse  :: k -> k -> STM k

deriving instance Functor STM
```

Figure 10: Our `STM` interface

Ideally, we would like that each operation has access to the `Heap` at all times, so that `New` corresponds to performing `alloc` on the heap, and passing the resulting `TVar` and the new heap into the continuation, `Read` correponds to performing `lookup` on the heap, and passing the resulting value of type `a` into the continuation, and `Write` corresponds passing

the modified heap from `update` to the rest of the computation. It is interesting to note that `OrElse` is parameterized by two continuations (transactions), which is essentially a way to contain the two alternative transactions in different scopes. Handlers can then evaluate their effects in isolation, and then decide with which continuation to proceed next. Additionally, in the single-threaded case, the `Retry` operation essentially becomes an aborting operation, since under no circumstances the transactional memory would be modified before the transaction's re-execution and the transaction would possibly end up in an infinite loop. By reinterpreting the semantics in this way, the behaviour of `Retry` and `OrElse` closely resembles that of `throw` and `catch` operations of the exception effect in Section 1. Taking all of this in mind, we arrive at our single-threaded handler implementation for the STM interface, statefully threading the `Heap` through the computation, as shown in Figure 7.

```
hSTM :: Functor f => StatefulHandler STM a Heap f (Maybe (a, Heap))
hSTM = StatefulHandler {
    retS = \a h -> return $ Just (a, h),
    hdlrS = \op h -> case op of
        New a k -> let (t, h') = alloc a h in k t h'
        Read t k -> let a = lookup t h in k a h
        Write t a k -> k $ update t a h
        Retry -> return Nothing
        OrElse t1 t2 -> t1 h >>= maybe (t2 h) (return . Just)
}
```

Figure 11: The single-threaded `STM` handler

3.2 Modelling Multi-Threading with Atomic Transactions

In order to actually model STM transactions running in parallel, we need a way to express and implement multi-threaded programs using algebraic effects. Wu et al. [20] proposed a concurrency effect modelling *cooperative multi-threading*, in which threads themselves relinquish control to the scheduler, so that it may decide to run a different thread instead. However, the framework of algebraic effects is not expressive enough to model this effect, as they shown it is a *higher-order effect*. In particular, `Fork w k`, which spawns a new worker thread `w` and continues with `k` on the main thread, is a scoped operation, and thus does not compose with traditional sequencing like algebraic effects: `Fork w k >>= f` $\neq$ `Fork (w >>= f) (k >>= f)`, but `Fork w k >>= f` = `Fork w (k >>= f)`

To that end, we adopt a framework for defining higher-order effectful programs called `Hefty` trees [2], which is a generalization of the free monad that supports higher-order effects (defined by higher-order functors), and allows their modular elaboration into algebraic effects. We borrow the code for the framework from [13]. Additionally, we extend these higher-order functors with the `weave` operation for ad-hoc handling of scoped effects directly on `Hefty` trees, as demonstrated by Wu et al., which greatly simplifies our scheduler implementation.

The signature higher-order functor of our `Thread` effect is an more of extension to the cooperative multi-threading, whilst discarding the `Yield` operation in favour of immediate yielding of the current thread after an operation finishes execution. Being able to declare operations like `atomic` and `wait` will prove essential in demonstrating our STM effect on a fully concurrent application in 5.1.

```
atomic  :: Thread <: h => Hefty h a -> Hefty h a
fork    :: Thread <: h => Hefty h a -> Hefty h ()
```

```
wait    :: (Alg Out <: h, Thread <: h) => Int -> Hefty h ()

execute :: (Alg Err <: h, Alg Out <: h)
    => Hefty (Thread :+: h) a -> Hefty h a
executeAll :: (Alg Err <: h, Alg NonDet <: h, Alg Out <: h)
    => Hefty (Thread :+: h) a -> Hefty h a
```

Figure 12: The `Thread` effect and its schedulers

In the Figure 12, the operation `atomic m` delimits a block `m` which is then executed all at once, i.e. no other thread starts running before it finishes execution. Its implementation also disallows any `Thread` operations (nested atomic blocks, forks and waits) to happen inside this block. The operation `wait d` suspends the the thread for `d` microseconds and then resumes the continuation (using an operation from `Out` [16]). The evaluation of concurrent programs can be modelled two ways:

- Using `execute` function — the program is run by a scheduler implemented similarly to the one from [20], it cycles through the currently being evaluated main thread (the input program) and the currently being evaluated worker threads (spawned with `fork` in the input program) in a round-robin manner. It switches from one thread to the other after a single (or atomic) operation is performed. The worker threads are kept track of in a two queues, the first queue is used to *pop* the currently running workers one by one, and the second queue is used to collect these workers to be evaluated in the next cycle by *pushing* the running workers' remaining computations. Additionally, the main thread does not terminate unless all worker threads terminate first. With this evaluation, it is possible to simulate a "real" scheduler, which proves useful when we demonstrate our STM interface on the example program in Section 5.

- Using `executeAll` function — every possible interleavings of the input program's main and worker threads using non-determinism. The switching between threads is now non-deterministic — every time a thread yields or forks, our model non-deterministically chooses a permutation of the current spawned workers ordering, where for each permutation it, again non-deterministically, chooses to either run the main thread or the first worker thread from permuted ordering. Just as with the previous evaluation scheme, the main thread terminates when all other worker threads have terminated first, which means all non-determinstic computations where the main thread terminates first are discarded. This evaluation scheme is useful when verifying whether our implementation computes the desired result for all thread interleavings of the given input program.

For more in-depth explanation of the semantics, we refer you to the code from this paper [16].

Since it is by definition a scoped effect, different composition with other effects yields different semantics. As example, Wu et al. mention the interaction with the `State s` effect, i.e. *local* state vs. *global* state semantics. When handling the state effect first, `fork` creates a *local* copy of the state within the individual thread and the changes to it are not shared. However, when it is the other way around, all threads interact with the *global* state and the changes are shared between them. Thus, the only way we can have parallel STM transactions validate and commit to the *shared* transactional memory at any point in the program's execution, is to handle the `Thread` effect first and then evaluate the logged changes to the global `Heap`. Unfortunately, these semantics (and also the actual

semantics of Haskell's STM in literature) introduce several drawbacks to our implementation of `atomically`:

- We cannot simply introduce the `STM` effect into the effect row of `Hefty`, as that would allow us to introduce `STM` operations outside of the `atomically` block.
- Since the optimistic synchronisation necessitates *isolation*, the transaction inside `atomically` has to be unaffected by other threads, i.e. it has to be executed against a snapshot of the transactional memory. Even though this behaviour corresponds to the transactional state interpretation of the effect interaction between state and exception catching from 1, with the scoped `Thread`, it is simply not possible to share the local updates with the main thread. Thus, we have to follow the global state interpretation.

The only option we are left with is to model the optimistic synchronisation by embedding the single-threaded handler from 3.1 into the implementation of `atomically` in Figure 13, at the cost of modularity.

```
atomically :: (Thread <: h, Alg (State Heap) <: h)
    => Free (STM + End) a -> Hefty h a
atomically t = do
    initial <- get'
    let eval = un $ handleStateful hSTM initial t
    case eval of
        Just (r, changed) -> do
            commit <- atomic $ do
                current <- get'
                if current == initial then
                    put' changed >> return True
                else
                    return False
            if commit then return a else atomically t
        Nothing -> atomically t
```

Figure 13: Our implementation of `atomically`

The global `Heap` would then be provided by the `State Heap` effect. The implementation exactly follows the optimistic synchronisation, in a sense that, the transaction is executed against the snapshot `initial`, which produces either the result paired with the changed heap `changed` or fails. If the transaction fails, we recursivelly call `atomically`, at which point the thread would automatically yield, and the scheduler would run another thread. If the transaction succeeds, the contents of `changed` are commited and the result returned if and only if the `initial` heap is consistent with/equal to the `current` heap, otherwise we re-run the transaction again. The commiting part is executed in a e interleaved with other thread which break consistency. In general, one might want to compose and execute these multi-threaded programs with STM transactions with type signature that yields the correct semantics, such as `Hefty (Thread :+: Alg (State Heap) :+: Alg Err :+: Alg Out) a`. In the Appendix A, we provide an example of a limited account withdrawal from [8].

4 Verifying the Model Implementation

With the implementation in place, the question arises as to how we can prove that our STM semantics follow a sound equational theory for STM abstractions. Such theory was

proposed by Borgström et al. [4], and allows reasoning about programs which use Haskell's `STM`. In this section, we outline correspondences between its formalisation of Haskell's `STM` expressions and our STM abstraction utilizing `Free` monads, and prove equivalences which should hold for our implementation for evaluating STM expression in `atomically`.

4.1 Core Calculus Correspondence

The core calculus for STM expressions from the given theory is defined as "a concurrent non-strict lambda calculus, with memory cells (`TVar`s) and atomic blocks (`atomically`)", so it does not concern the specifics of the concurrency model of `IO` monad, except that it allows parallel execution of transaction. Thus it is not necessary to verify whether our implementation of the `Thread` effect follows any specific process calculus laws, since the theory does not provide any, and because it is also outside of the scope of this paper). For the purposes of our implementation, we are interested only in specifying the single-threaded handling behaviour of our `handleSTM` function, on which `atomically` relies on.

As given by the formal definition by Borgström et al., (single-threaded) STM expressions have the form of $H \mid M$ which is a parallel composition of a heap of transactional variables H and a running transaction M. Moreover, the operational semantics of operations in M do not consist parallel compositions with the full heap, but only with an adjacent piece of the heap with which we actually perform the transaction. In essence, each transaction has to be accompanied with a view of the memory (heap) shared with other transactions in the program. The equivalence between two transaction expressions M and N is established succinctly using heap transformer equivalence ($\leftrightarrow$). Two STM expressions satisfy such equivalence if and only if for any arbitrary heap both expression reduce to the *same result* or the *retry operation* together with *structurally equivalent* heaps.

In comparison with our implementation, the operational semantics are modelled by evaluating `Free (STM + f) a` transactions with `handleSTM`. Although the equivalences concern isolated `STM` expressions, the functor `f` may possibly introduce other effects. Therefore, we consider `Free` expressions where we take `f` to be the `End` effect, as it does not have any constructors, and thus does not introduce other arbitrary syntax into the expressions. We now have a correspondence with the type system, such that expressions $\texttt{>>=}_{\text{STM}}$ and $\texttt{return}_{\text{STM}}$ are equivalent to $\texttt{>>=}_{\text{Free}}$ and $\texttt{return}_{\text{Free}}$. Additionally, `handleSTM`'s inner workings necessitate a view of the heap threaded in its computation, which is either returned along with the result (commit) or discarded (retry) in terms of `Maybe`. This behaviour actually corresponds to the formalism of STM expressions and their parallel composition with the heap. Therefore, in a single-threaded case, it is possible to establish heap transformer equivalence between two transactions, as when we have means of comparing the heap after execution and the outcome using equational reasoning. Proving that $M \leftrightarrow N$ is now analogous to evaluating `handleSTM` M $h =$ `handleSTM` N h for arbitrary heap h.

4.2 Proofs of Imperative Equivalences and Properties of Operations

We focus on outlining the proofs of the necessary equivalences our implementation has to satisfy using the established reasoning technique. These are summarized in Lemma 15 and 16 in the paper by orgström et al. Since the proofs are too extensive to put into this paper, we include them with the code of this paper [16] which includes also the proofs of these equivalences:

1. $(\texttt{readTVar}\ a\ \texttt{>>=}_{\text{STM}}\ \lambda x.\texttt{writeTVar}\ a\ x) \leftrightarrow \texttt{return}_{\text{STM}}()$
2. $(\texttt{writeTVar}\ a\ M\ \texttt{>>}_{\text{STM}}\ \texttt{writeTVar}\ b\ N) \leftrightarrow$
   $(\texttt{writeTVar}\ b\ N\ \texttt{>>}_{\text{STM}}\ \texttt{writeTVar}\ a\ M)$ if $a \neq b$
3. $(\texttt{readTVar}\ a\ \texttt{>>=}_{\text{STM}}\ \lambda x.\texttt{writeTVar}\ b\ M\ \texttt{>>=}_{\text{STM}}\ \texttt{return}_{\text{STM}}\ x) \leftrightarrow$
   $(\texttt{writeTVar}\ b\ N\ \texttt{>>}_{\text{STM}}\ \texttt{readTVar}\ a)$ if $a \neq b$
4. $\texttt{orElse}\ \texttt{retry}\ M \leftrightarrow M$
5. $\texttt{orElse}\ M\ \texttt{retry} \leftrightarrow M$
6. $\texttt{orElse}\ M_1\ (\texttt{orElse}\ M_2\ M_3) \leftrightarrow \texttt{orElse}\ (\texttt{orElse}\ M_1\ M_2)\ M_3$

5 Applications

This section elaborates on the uses of our STM model in different applications, namely translating an existing Dining Philosophers implementation using Haskell's STM into a multi-threaded `Hefty` program, and extending the notion of transational memory for other algebraic effects.

5.1 Dining Philosophers

The Dining Philosophers[3] problem is a classic exercise in concurrent programming which illustrates that concurrent synchronisation primitives, like lock-based semaphores and mutexes, do not compose well. This is precisely the issue which STM tries to alleviate, which makes solving the problem relatively straightforward. We found a solution[4] to the dining philosophers problem, which uses Haskell's STM, that we decided to translate into our framework of algebraic effects. It turns out that by simply changing the type signatures of the provided functions from monadic effects (`STM` and `IO`) to either `Free` or `Hefty`, we can essentially reinterpret the solution whilst keeping the syntax and the functionality intact, apart from implementing `TMVar`s[5]. The adapted solution is supplied with the code to this paper [16].

5.2 Extending Transactional Memory For Other Effects

Although embedding the single-threaded handler in the syntax `atomically` certainly diminishes the modularity of `STM` as an algebraic effect, it is still possible to generalize this encoding for any effect that might follow the same transactional semantics.

The goal is to model different effects as transactions acting on a transactional memory, possibly allowing operations `retry` and `orElse`. The transactional memory in the case of `STM` was a `Heap`, which was passed along and returned with the result in the parameterized handler `hSTM`. We can surmise that if we can implement a parameterized handler passing around and returning a parameter `s` for a particular effect whose set of operations act on this parameter, then `s` can represent the transactional memory for that effect, just as `Heap`

[3] `https://en.wikipedia.org/wiki/Dining_philosophers_problem`

[4] `https://rosettacode.org/wiki/Dining_philosophers%23Haskell`

[5] `TMVar`s are a transactional synchronising primitive used for communication between threads(e.g. signaling resource availability) `https://hackage.haskell.org/package/stm-2.5.3.1/docs/Control-Concurrent-STM-TMVar.html`

represented the transactional memory for our STM effect. Additionally, the effect in question does not need to implement retry and orElse operations in its interface. We can decouple these operations from STM to a different effect which we call Transactional, as shown in Figure 14.

```
data Transactional k where
    Retry :: Transactional k
    OrElse :: k -> k -> Transactional k
    deriving Functor

retry           :: Transactional < f => Free f a
orElse          :: Transactional < f => Free f a -> Free f a -> Free f a
hTransactional  :: Functor f => Handler Transactional a f (Maybe a)
```

Figure 14: Our implementation of atomically

The semantics of hTransactional are the same as the semantics of hSTM for Retry and OrElse. Thus, we now have the means of constructing a generalized version of atomically, which we name atomically' (Figure 15).

```
atomically' :: (Eq m, Functor f, Thread <: h, Alg (State m) <: h)
    => (forall f'. Functor f' => StatefulHandler f a m f' (a, m))
    -> Free (f + Transactional + End) a -> Hefty h a
```

Figure 15: Extended version of atomically to other effects

The implementation of atomically' is practically identical to atomically, with the addition of passing the StatefulHandler to specify the relationship between the effect f and its transactional memory m, but isolates the effect. We evaluate the effect with Transactional by first applying the StatefulHandler with the current view of the transactional memory m and then applying hTransactional, where we get either the result paired with the changed memory or a failed computation. Afterwards, we proceed as with the atomically for STM. Thus, we do not introduce any duplicate effects from f into the Hefty program, as it is not present in its effect row. Moreover, m needs to be an instance of Eq for the consistency check. In the Appendix B, we provide an example of extending the Writer effect.

6 Responsible Research

In this section, we discuss the measures undertaken in our study, in order to conduct research that aligns with responsible and ethical principles. A particular attention was given to ensure our research is both reproducible and transparent.

The code from this paper is available freely on GitHub [16], which we refer to throughout our work multiple times. It includes the entire framework for Free monad, Hefty trees, our STM interface and its handler, Thread effect with its schedulers, and everything else from Sections 2, Section 3 and Section 5.

Furthermore, we constructed proofs about the operations of our interface that ensure correctness of our implementation in Section 4. In the context of STM transactions, incorrect program behaviour could lead to runtime errors, the corruption of shared data and race conditions. Unfortunately, for the sake of brevity, they could not be included in the paper. However, they are included with the code on GitHub as well, so that others may easily verify them.

Following the responsible research practices such as reproducibility and transparency, our work aims to promote responsible practices in the field.

7 Discussion

In this section, we critically assess the proposed implementation of STM in our work, its advantages and limitations. We are particularly interested in exploring possibilities for improving our current model, and whether it is viable to implement Haskell's STM abstraction in terms of algebraic effects and handlers.

By composing our STM interface with other effects in sequential programs in Section 3.1, we have found their effect interaction resembling the transactional semantics between state and exception catching from Section 1. However, this interpretation was not possible to replicate when the interface was brought into a concurrent setting. The main reason for this difference is *re-execution*, a concept specific to STM only. In single-threaded programs, our implementation interprets `retry` operation as an aborting operation and `orElse` operation as a catching operation. These programs can be thought of as being separate transactions (possibly containing other effects), which makes them not particularly useful as a model, since they do not respect the proper semantics of Haskell's STM, and that STM is primarily a *concurrency* abstraction. Therefore, we introduced a simple concurrency model using scoped effects in which it is possible to use shared (global) mutable state between different threads, and hence we are able to implement re-execution in `atomically`. In this case, our implementation cannot rely on transactional semantics, because they simply do not emerge from the interaction our concurrency effect has with other effects, and because no STM operations cannot be executed outside `atomically`, as explained in Section 3.2.

These restrictions limit our implementation in terms of modularity that makes algebraic effects so attractive. To answer the research question within the given timeframe. With our concurrency model, our only option was to embed the single-threaded `STM` handler in order to evaluate the transaction block and provide correct optimistic synchronisation semantics in `atomically`. We can argue that this approach is modular in a sense that it can be extended to other effects by embedding their own handlers to `atomically` (see Section 5.2), however, coupling syntax (operations) with implementation (handlers) defeats the purpose algebraic effects and handlers. As there is striking lack of models and literature regarding integration of transactional memories into algebraic effects and handlers, without depending on ad-hoc solutions, we propose that it is explored further as part of the future development.

8 Related Work

Algebraic Effects and Handlers For many years, since their introduction into functional programming languages [7], *monads* have been the most dominant approach for modelling computational effects. However, they were notoriously difficult to compose, which lead to the development of *monad transformers* [6]. Plotkin and Powell [10] were then the first to develop an *algebraic* approach to effects whose operations follow a set of equations, offering higher modularity than monad transformers. Later Plotkin and Pretnar later extended this idea with the concept of *handlers* [11] to support handling exceptions.

Higher-Order Effects One of the more closely related work to this paper is the work of Wu et al. [20], introducing the notion of *scoped effects* and providing us with an implementation

of a concurrency model based on *cooperative multi-threading*. In our work, we extend their concurrency model with atomic operations, automatic relinquishing of control and tracing all possible interleavings of threads using non-determinism. Apart from their ad-hoc approach to scoped effects, Bach et al. [2] provided a general encoding of higher-order effects, including scoped and latent effects [18], using their so-called hefty algebras, which we adopt in this paper as well. Instead of handling these effects, they provide their elaborations into algebraic effects. Even more general framework was explored by van der Berg and Schrijvens [17], which we have not yet investigated, although it might be an interesting structure to apply in the future work.

Software Transactional Memory STM in Haskell [8] [5] as concurrency abstraction was developed to alleviate previous short-comings of Concurrent Haskell [9] with atomic transactions, and has been the guiding resource for our implementation. Later, there have been explorations of formal calculus for Haskell's STM by Acciai et al. [1], however these did not provide any equational theory for Haskell code, nor any significant examples. Their work was superceded by Borgström et al. [4].

Unfortunately, there seems to be a lack of research in the field of integrating STM into algebraic effects, which makes our approach quite novel. However, Punchihewa and Wu [14] have attempted to implement safe mutation in concurrent programs using algebraic effects, however, their focus is on region-based memory management rather than transactional memory.

9 Conclusions and Future Work

In our work, we first implemented an `STM` interface and its handler (`hSTM`) for sequential (single-threaded) `Free` programs [12], modelling the operations of Haskell's STM [8] [5]. Then we provided an implementation of `atomically` that embeds the `STM`'s single-threaded handler into the syntax of concurrent programs using higher-order effects [13] [2] [20], which emulates optimistic synchronisation of Haskell's STM. Additionally, we provided formal proofs that the implementation satisfies the necessary laws and equivalences for the operations of Haskell's STM transactions [4]. Finally, we demonstrated our implementation on an example solution of the dining philosophers problem using Haskell's STM transactions, and extended the notion of concurrent atomic transactions and transactional memory to other (stateful) effects by implementing their `atomically`-equivalent operation — `atomically'`. With these contributions made, we have answered all of our research questions.

For future work, it should be explored how to decouple the single-threaded handler from the syntax of `atomically`, as it breaks the separation of syntax and semantics. Moreover, composing multiple effects in the transaction block that can be effectively re-executed without accidental duplication of these effects also remains to be investigated, as our implementation allows execution of only a single effect in `atomically`.

An additional development for future work would be extending our interface with `throw` and `catch` actions for STM [5]. Furthermore, extending our current concurrency model such that it prevents transactions that block their thread indefinitely, e.g. transactions consisting of a single `retry` operation. Also regarding the concurrency model, there is lack of any formal reasoning regarding the behaviour of `atomically` operation, since the resource we used to prove laws about our `STM` interface [4] does not provide equivalences which hold of this operation, which might be an interesting piece of future work to explore.

References

[1] Lucia Acciai, Michele Boreale, and Silvano Dal Zilio. A Concurrent Calculus with Atomic Transactions, 2006. arXiv:cs/0610137.

[2] Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571255.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 01 2015. URL: http://dx.doi.org/10.1016/j.jlamp.2014.02.001, doi:10.1016/j.jlamp.2014.02.001.

[4] Johannes Borgström, Karthikeyan Bhargavan, and Andrew D. Gordon. A compositional theory for STM Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 69–80, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596638.1596648.

[5] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. PPoPP '05, pages 48–60, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1065944.1065952.

[6] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. Association for Computing Machinery. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/199448.199528, doi:10.1145/199448.199528.

[7] E. Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989. doi:10.1109/LICS.1989.39155.

[8] Simon Peyton Jones. *Beautiful concurrency.* O'Reilly, beautiful code edition, 01 2007. URL: https://www.microsoft.com/en-us/research/publication/beautiful-concurrency/.

[9] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 11 1995. doi:10.1145/237721.237794.

[10] Gordon Plotkin and John Power. Algebraic Operations and Generic Effects. *Applied Categorical Structures*, 11(1):69–94, 2003. URL: http://dx.doi.org/10.1023/A:1023064908962, doi:10.1023/a:1023064908962.

[11] Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, 12 2013. URL: http://dx.doi.org/10.2168/LMCS-9(4:23)2013, doi:10.2168/lmcs-9(4:23)2013.

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

[13] Casper Bach Poulsen. Algebras of Higher-Order Effects in Haskell, 08 2023. URL: http://casperbp.net/posts/2023-08-algebras-of-higher-order-effects/.

[14] Hashan Punchihewa and Nicolas Wu. Safe mutation with algebraic effects. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, pages 122–135, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3471874.3472988.

[15] Wouter Swiestra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008. doi:10.1017/S0956796808006758.

[16] Matej Tomášek. Algebraic Effects and Handlers for Software Transactional Memory, 2024. URL: https://github.com/watc4d0gg/Algebraic-Effects-and-Handlers-for-Software-Transactional-Memory.

[17] Birthe van den Berg and Tom Schrijvers. A framework for higher-order effects handlers. *Science of Computer Programming*, 234:103086, 2024. URL: https://www.sciencedirect.com/science/article/pii/S0167642324000091, doi:10.1016/j.scico.2024.103086.

[18] Birthe van den Berg, Tom Schrijvers, Casper Bach-Poulsen, and Nicolas Wu. Latent Effects for Reusable Language Components: Extended Version, 2021. arXiv:2108.11155.

[19] Cas van der Rest, Jaro Reinders, and Casper Bach Poulsen. Handling Higher-Order Effects, 2022. arXiv:2203.03288.

[20] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect Handlers in Scope. In *Proceedings of the 2014 Haskell Symposium*, Haskell '14, New York, NY, USA, 2014. ACM. URL: http://www.cs.ox.ac.uk/people/nicolas.wu/papers/Scope.pdf.

Appendix A: An Example with Limited Withdrawal

```haskell
runProg :: Show a => Hefty (Thread :+: Alg (State Heap) :+: Alg Err :+: Alg Out) a -> IO ()
runProg prog = do
    r <- io
        $ handleError
        $ handleState ([] :: Heap)
        $ hfold Pure (eAlg /\ eAlg /\ eAlg)
        $ execute prog
    print r

transactTest :: IO ()
transactTest = runProg $ do
    acc <- atomically $ do newTVar 0
    fork $ atomically $ limitedWithdraw acc 42
    fork $ atomically $ writeTVar acc 1337
    where
        limitedWithdraw :: STM < f => TVar Int -> Int -> Free f ()
        limitedWithdraw acc amount = do
```

```
            bal <- readTVar acc
            check (amount <= 0 || amount <= bal)
            writeTVar acc (bal - amount)

> Right ((),[Cell 1295])
```

Appendix B: An Example Extension of the Writer effect

```
data Writer m k where
    Tell :: m -> k -> Writer m k
    deriving Functor

tell :: Writer m < f => m -> Free f ()
tell m = Op $ inj $ Tell m (Pure ())

hWriter :: (Monoid m, Functor f) => StatefulHandler (Writer m) a m f (a, m)
hWriter = StatefulHandler {
    retS = curry pure,
    hdlrS = \(Tell m' k) m -> k $ m <> m'
}

atomicallyWriter :: (Eq m, Monoid m, Thread <: h, Alg (State m) <: h, Alg Out <: h)
    => Free (Writer m + Transactional + End) a -> Hefty h a
atomicallyWriter = atomically' hWriter
```