

# An Implementation and Semantics for Transactional Memory Introspection in Haskell

Reykjavik University Technical Report

RUTR-CS08007

# An Implementation and Semantics for Transactional Memory Introspection in Haskell

Arnar Birgisson   Úlfar Erlingsson  
Reykjavik University

October 28, 2008

**Note:** This is a draft version of Reykjavik University Technical report RUTR-CS08007.

## Abstract

Transactional Memory Introspection (TMI) is a novel reference monitor architecture that provides complete mediation, freedom from time-of-check-to-time-of-use bugs and improved failure handling for authorization, by building on the Software Transactional Memory architecture. In this paper we present a formal definition of TMI and a concrete implementation over the Haskell STM.

The first author was partly supported by the project “New Developments in Operational Semantics” (nr. 080039021) of The Icelandic Research Fund.

## 1 Introduction

The implementation of security enforcement mechanisms requires special care, as any flaw may open the door to malicious attacks. This is especially true in the case of multithreaded software, as the designer must consider all possible interleavings of code paths. In [1] we presented Transactional Memory Introspection (TMI), an architecture that greatly simplifies the implementation of correct reference monitors on Software Transactional Memory (STM) architectures.

STM provides many useful guarantees that make the implementation of multithreaded software easier and less error-prone. The mechanisms of STM provide among other things atomicity and isolation. This is accomplished by optimistically executing concurrent code and monitoring for conflicting accesses to resources. By providing rollback semantics, STM resolves conflicting accesses by undoing work by transactions and retrying them from the top.

All STM implementations must perform bookkeeping of accesses (such as reads and writes) to shared resources. By interposing on this bookkeeping and the monitoring and validation steps, TMI provides the designer with the facilities to implement robust and correct enforcement mechanism. TMI provides complete mediation by requiring a separate authorization check before committing a transaction, in addition to the concurrency conflict check. TMI also simplifies error handling. When unauthorized accesses are detected in a transaction, it is rolled back and not retried. This saves the programmer from performing clean-up after a failed authorization.

Another common problem in traditional authorization is *time of check to time of use* bugs. This happens when an authorization check either grants or denies an operation, but interleaving of code may change some state so that when the time comes to perform the operation, the authorization decision has become invalid. By executing both the check and the operation inside a transaction, solely using STM eliminates this problem.

In [1] we provide a comprehensive overview of the TMI architecture and evaluate an implementation in Java, building on the DSTM2 library [3]. In order to provide clear, well-defined semantics for TMI, this paper gives the topic a more formal treatment and defines structural operational semantics for the TMI architecture. For this we need

an STM system with equally rigid semantics. This is provided by the Haskell STM, whose semantics are given in [2]. Our semantics of TMI builds on this foundation and is accompanied by an implementation over the Haskell STM system as well.

## 2 Background

### 2.1 STM and TMI

STM provides attractive guarantees for multithreaded software, namely atomicity, consistency and isolation of *transactions*, specifically marked blocks of code. Most current implementations of STM do so by performing

- careful monitoring of accessed resources within a transaction,
- validation the acceses of concurrent transactions, and
- complete rollback of the effects of aborted transactions.

TMI builds on this machinery and allows security enforcement to benefit from the STM guarantees. TMI helps the programmer to write correct enforcement mechanisms and simplifies error-handling. In [1] we outline three main benefits of TMI.

**Complete mediation.** TMI provides complete mediation by implicitly invoking the reference monitor before any effects of a transaction are permanently committed. The reference monitor is able to inspect the resource access logs of the STM and may veto the commit if an application specific policy is violated. This depends partly on the STM providing strong atomicity, i.e. resources marked for transactional scrutiny may not be accessed outside the scope of a transaction.

**Freedom from TOCTTOU bugs.** Time-of-check-to-time-of-use (TOCTTOU) bugs arise in conventional enforcement mechanisms when interleaved threads may affect the policy decisions of each other. For example, a thread may make a policy-based decision to allow access to a certain resource, e.g. reading a memory location. Before that operation is actually performed, execution may be preempted by another thread. That thread can change the global state so that the policy decision becomes invalid, e.g. by writing privileged information into the memory location.

This problem is implicitly solved by using STM, which guarantees that transactions are completely isolated and cannot affect each other.

**Simplified error handling.** In the event of an authorization failure TMI uses the STM facilities to completely roll back the effects of the transaction in question and raises an appropriate exception to the code that initiated the transaction. This frees the programmer from having to undo state changes leading up to the unauthorized operation, a common source of errors [].

### 2.2 Haskell STM

For a formal treatment, we build our semantics and implementation on that of the Haskell STM [], which in turn is built on Concurrent

Haskell [7]. Concurrent Haskell is an extension to Haskell 98, a lazy (i.e. call-by-name), pure, functional language. It supports concurrent threads and communications between them. Side-effects are, as generally in Haskell, handled with *monads* [5].

At the core of a Haskell program is the type **IO a**, representing an I/O action that possibly performs some I/O and then returns a value of type *a*. Due to Haskell's type system, such actions may be composed to control order of evaluation in an otherwise lazy environment. The type system also ensures that I/O actions are contained and cannot have an effect on the evaluation pure functions. Any complete Haskell program defines a single I/O action called `main`, which serves as its main entry point.

Composition of I/O actions happens through a *monadic bind* operator, often used through syntactic sugar that resembles an imperative setting. [2] provides this example of a program that reads a character and then prints it twice,

```
main = do { c <- getChar; putChar c; putChar c }
```

In this case, three I/O actions are combined to create the main I/O action.

In addition to conventional input and output, I/O actions can perform reads and updates of mutable cells. The type `IORef a` represents a mutable cell that contains a value of type *a*. Cells may be updated only within the context of I/O actions through an interface limited to creating, reading values from, and writing values into cells. Manipulation of these cells cannot happen in pure code, only in I/O actions.

Concurrent Haskell supports explicit forking of threads through the function `forkIO`.

```
forkIO :: IO a -> IO ThreadID
```

`forkIO` takes an I/O action as a parameter and spawns a new thread to execute the action, immediately returning a newly allocated thread identifier. For further discussion of concurrency we refer to [4] or tutorials such as [].

The Haskell STM [2] also uses the type system to limit operations on *transactional variables* to specific code, namely *STM actions*. STM actions have a type similar to I/O actions, `STM a`. This type represents a computation that performs operations on transactional variables and returns a value of type *a*. Transactional variables are similar to the mutable cells of I/O actions. Their type is `TVar a`, representing a variable containing a value of type *a*. Functions for creating, reading and writing such variables have their return type wrapped in the STM monad, which means these operations can only be performed as a part of an STM action.

STM actions can be composed. Similar to I/O actions, the monadic bind operator ties them together to be performed in sequence, but in addition Haskell STM provides the primitive action `retry` and a combinator `orElse`. `retry` is a blocking operation for STM actions. By issuing it, the programmer is stating that the current transaction cannot finish for the current state of transactional variables. Haskell STM uses the read and write sets of the transaction to note which variables were accessed by it, rolls it back and retries it from the top when any of these variables change.

If `t1` and `t2` are STM actions, then `t1 `orElse` t2` is an STM action that first tries performing `t1` on its own. If `t1` evaluates to the `retry` action, then the combined action rolls back the effects of `t1` and tries `t2` instead. If that one retries, the whole action does so also, but waits for updates on the variables read by *both* `t1` and `t2`.

For an example how the above can be used for synchronization primitives such as communication channels, see section 4 of [2].

The above gives us a way to define STM actions, units of work that can work with transactional variables and perform pure computations. With such an action in hand, we can trigger its execution

in the program's I/O context through the `atomically` function<sup>1</sup>. This function has the type

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

and takes an STM action and returns an I/O action, that when performed, will run STM action atomically with respect to concurrently running STM actions. `atomically` is the only gateway from the world of STM actions to the world of I/O actions and thus the main program. As an example, the following program creates a transactional variable holding a counter and spawns three threads that each increments the counter atomically.

```
increment :: TVar Int -> STM ()
increment counter = do x <- readTVar counter
                      writeTVar counter (x + 1)
```

```
main = do c <- atomically (newTVar 0)
          forkIO (atomically (increment c))
          forkIO (atomically (increment c))
          forkIO (atomically (increment c))
```

### 3 Transactional Memory Introspection

In this section we give an overview of the TMI architecture and how it is implemented. We then describe our Haskell implementation from a user standpoint.

#### 3.1 Overview of TMI

As described in our previous paper [1], the TMI architecture aims to raise the level of abstraction in the implementation of security enforcement mechanisms. It allows the programmer to decouple application logic from security enforcement. Just as STM frees the programmer from worrying about lock acquisition order and other synchronization efforts, TMI eliminates concerns about check placement, race conditions and exceptional execution paths.

TMI does this by imposing on the STM system. The programmer marks certain variables as security sensitive. This implicitly marks them as shared, meaning the STM system will start to protect access to them from race conditions. TMI adds on that by monitoring access to them as well, invoking an application specific reference monitor every time they are accessed. In [1] we mainly two possible implementations of this architecture, *eager* and *lazy enforcement*. In *eager* enforcement, every access to a variable triggers the reference monitor, which immediately checks it against the relevant policy. If authorization is denied, the transaction is immediately aborted. In *lazy enforcement*, accesses to variables are simply logged (often they are already logged by the STM) and only at the end of the transaction, the logs are inspected by the reference monitor. If any of the logged accesses are invalid, the whole transaction is aborted. Their key point here is that we only need to make the policy decision before the transaction is committed. A good STM system will ensure that execution within a transaction are not observable and aborting one will have the same semantics as not performing it at all. This is why every operation during the transaction is fair game, as long as it is authorized before the final commit. For our formal treatment and Haskell implementation, we focus on *lazy enforcement* only. Thus, the following discussion only deals with the *lazy* variant unless otherwise noted.

To use TMI, the programmer declares a set of variables as security relevant. The variables are implicitly registered with the STM system as shared as well. This means the values held by these variables can only be read or modified within a transaction, and that the STM

<sup>1</sup>While [2] uses `atomic`, the actual implementation of the Glasgow Haskell Compiler uses `atomically`.

system takes care of resolving conflicting accesses by concurrent transactions. Upon every variable access, in particular the creation, reading or writing of a variable, TMI enters information identifying the variable in question to a transaction-specific *introspection log*. The programmer then marks specific sections of code, containing accesses to the variables, as *atomic*. To execute these blocks, she initiates a transaction, providing a reference to the atomic block and a *security manager*. The security manager is a block of code (or closure) that is responsible for looking at the access log of a transaction and deciding if it satisfies the application specific policy or not. The security manager must contain any auxiliary information it needs to make this decision, such as the active principal etc. The TMI system then initiates a transaction in the STM system that starts by running the atomic block. After that block finishes, TMI extracts the access log of the transaction at that point and passes it to the security manager, still running within the bounds of the transaction. If the security manager returns success, the transaction ends and the STM system performs its regular validation to see if it may be permanently committed or if it needs to be retried. If the security manager however indicates failure, TMI instructs the STM system to abort the transaction and instead raises an exception to the code that initiated it. As a simple example, consider the following pseudo-code, where we allow currying of function arguments.

```

declare sensitive accounts = array of Account

function withdraw(account, amount):
    account.balance = account.balance - amount

function security_manager(user, log):
    if log contains <withdrawal from account>:
        if account.owner == user:
            return Allowed
        return Denied

main program:
    user = acquire_login_credentials()
    try:
        transaction with security_manager(user):
            withdraw(get_account(123456), 42)
    catch AuthorizationFailed:
        tell user about error

```

There are two things to note here. First, the function `withdraw` performs no authorization. The security enforcement code is completely decoupled from the application logic. Second, in the case of an authorization failure (e.g. if the logged in user does not own account no. 123456), then the error handler for it only needs to worry about indicating the error to the user, as any state changes that happened during the transaction have already been rolled back. In other words, there is no mess for it to clean up. The benefits may seem insignificant for this simplified example, but for more complex operations composed of several security critical operations, they should be fairly obvious. These two observations, along with the freedom from TOCTTOU bugs are what we consider the main benefits of TMI. An example of a TOCTTOU here would be if accounts could change owners. Such an update would be isolated from other transactions and so the withdrawal and the access check would be guaranteed to see the same owner.

### 3.2 TMI in Haskell

We saw earlier how Concurrent Haskell uses the type system to confine operations on shared variables to STM actions, and provided a single function to wrap STM actions into an atomic I/O action. For

TMI, we do something very similar. We confine operations on security sensitive variables to *TMI actions*, and provide a single function to turn a TMI action into an STM action and associating it with a security manager at the same time.

	$x, y$	$\in$	Variable
	$r, t$	$\in$	Name
	$c$	$\in$	Char
Value	$V$	$::=$	$r \mid c \mid \backslash x \rightarrow M$ $\mid \text{return } M \mid M \gg=, N$ $\mid \text{putChar } c \mid \text{getChar}$ $\mid \text{throw } M \mid \text{catch } M N$ $\mid \text{retry} \mid M \text{ `orElse` } N$ $\mid \text{forkIO } M \mid \text{atomically } M$ $\mid \text{newTVar } M$ $\mid \text{readTVar } r \mid \text{writeTVar } r M$ $\mid \text{newTMIVar } N M$ $\mid \text{readTMIVar } r \mid \text{writeTMIVar } r M$ $\mid \text{authorized } N M \mid \text{liftSTM } M$ $\mid \text{getlog} \mid \text{UnauthorizedError}$
Term	$M, N$	$::=$	$x \mid V \mid M N \mid \dots$

Figure 1: Syntax of values and terms

Figure 1 shows the extensions of STM Haskell with the TMI extensions (highlighted). We define a new monad that represents TMI actions and operations on sensitive variables. In addition, we lift all standard STM functions to their TMI counterparts. This is done so that existing program using STM can be easily adapted to TMI with minimal changes to code. The TMI monad also encapsulates state, namely the introspection log of a transaction. The introspection log contains entries which specify the access type (create, read or write) of a variable and the *security descriptor* of a variable. Variable descriptors are provided by the programmer when she creates sensitive variables and contain variable metadata necessary for authorization, such as the owner of an account, permissions of a file, etc.

Since the type of security descriptors is applications specific, our new monad type is polymorphic,

TMI d a

where d is the type of descriptors and a is the type returned by the action. For security sensitive variables, we have a type similar to `IORef a` and `TVar a`,

TMIVar d a

An instance of this type is a cell with a security descriptor of type d and a value of type a. While the value can change over time, the security descriptor is specified when the cell is created and cannot change after that. Creation, reading and writing of cells is performed with the following set of functions.

```

newTMIVar :: d -> a -> TMI d (TMIVar d a)
readTMIVar :: TMIVar d a -> TMI d a
writeTMIVar :: TMIVar d a -> a -> TMI d ()

```

For an example, the following code defines a descriptor type for a bank account. The account itself is represented by a simple integer.

```

data AccountDescriptor = AccountDescriptor {
    accountOwner :: String,
    accountNumber :: Int
}
type Account = TMIVar AccountDescriptor Int

```

```
createAccount :: String -> Int -> Int -> TMI Account
createAccount owner number balance =
  newTMVar (AccountDescriptor owner number) balance
```

The next function demonstrates reading and writing variables.

```
deposit :: Account -> Int -> TMI AccountDescriptor ()
deposit acct amount =
  do balance <- readTMVar acct
     writeTMVar acct (balance + amount)
```

To turn a TMI action into an STM action, we need to associate it with a security manager, i.e a boolean function that evaluates the log of accesses and determines if the transaction should be aborted. As an input to this function, TMI defines the type of an access log.

```
data AccessType = CreateVar | ReadVar | WriteVar
type TMILog d = [(AccessType, d)]
```

To specify the application specific policy, the programmer must supply the security manager, a function of the type `TMILog d -> Bool`. This function, along with a TMI action is passed to the `authorized` function. The simplest security manager is one that performs no authorization and simply allows all operations.

```
allowAll :: forall d. TMI d a -> STM a
allowAll tx = authorized (const True) tx
```

A slightly more complex example is a security manager that looks at all Accounts touched by a transaction and verifies that they belong to the current user. The current user is passed to the security manager as the first argument, thus we can get the type needed for `authorized` by partially applying it.

```
sm :: String -> TMILog AccountDescriptor -> Bool
sm user thelog = all checkowner thelog
  where
    checkowner :: LogEntry AccountDescriptor -> Bool
    checkowner (_,descr) = user == (accountOwner descr)

-- authorized :: (TMILog d -> Bool) -> TMI d a -> STM a
main = do acct <- atomically (allowAll mkAccount)
      atomically (mkDeposit acct "alice") -- succeeds
      atomically (mkDeposit acct "bob")   -- fails
  where
    mkAccount = createAccount "alice" 123456 0
    mkDeposit acct user =
      authorized (sm user) $ deposit acct 42
```

Since TMI actions are ultimately lifted to STM actions, we also provide a lifting operation to lift STM operations into TMI operations, `liftSTM`. This allows for the embedding of an STM action inside a TMI action. Once the TMI action is turned into an STM action via `authorized`, the embedded action is just composed with it in the normal way. An interesting effect of this is that it allows for nested calls to `authorize`. While this may seem dangerous at first, we believe the semantics for this are clear and well defined and see no reason to prevent it.

TMI actions are also composable in the same way STM actions are. This means the monadic `bind` acts as sequential composition and we provide `orElseTMI` and `retryTMI` that behave as their STM counterparts. When TMI actions composed with `orElseTMI` are turned into STM actions via `atomically` the security manager only sees the log entries for `TMVar`-actions that are actually committed or could have affected the committed actions.

## 4 Formal semantics of TIM

To formalize the semantics of TMI, we build on the semantics for the Haskell STM presented in [2]. The semantics are structural operational semantics in the style of Plotkin [6]. For sake of completeness and to help the reader understand our extensions, we give a cursory explanation of the concepts of the semantics from [2] so that a reader not familiar with it may understand our extensions.

Figures 2 and 3 give the operational rules that describe the steps a program may take. At the top level, a program transforms a state of the form  $P; \Theta$  via labelled transition.

$$P; \Theta \xrightarrow{a} Q; \Theta'$$

$P$  represents a program term in the syntax of figure 1 while  $\Theta$  stands for a memory store, a partial function from variable names to annotated terms. An annotated value is a tuple  $(t, d)$  where  $t$  is a program term and  $d$  is a value that holds the security-relevant description of the relevant variable. The labels on transitions represent the programs input and output actions.  $Q$  and  $\Theta'$  represent the term that is left unevaluated and the updated store after a transition, respectively.

To model atomicity of transactions, separate relations represent the top level I/O transitions and the STM actions. We extend this by adding a third relation representing the security relevant TMI actions. Furthermore we add a simple relation for evaluation of security managers under the context of an immutable transaction log.

Execution of a program proceeds by non-deterministically picking a program term from a collection of terms, each representing a separate thread of execution. One I/O transition of this term combined with the current store is performed and then the process is repeated. This models interleaved concurrency at the level of I/O transitions. STM transitions however can only be performed as a required premise of the `atomically` operator at the I/O level, and thus appear in this model as a single atomic step.

As mentioned in [2] there is no need to represent rollback, but contrary to their semantics, our extensions do need to formalize the notion of the transaction log as it is no longer purely an implementation detail. For simplicity though, we only model the log for security sensitive operations as they are the only ones relevant to the semantics of TMI.

### 4.1 Syntax, states and evaluation contexts

The syntax of terms for a subset of STM Haskell is given in figure 1 with our TMI-related extensions (highlighted). Terms and values are conventional except that the application of some monadic operators are considered values, a technique again lifted from [2]. Also conventional, but included for sake of completeness, the `do`-notation used up until now is syntactic sugar for `return` and `>>=`.

$$\begin{aligned} \text{do } \{x \leftarrow e; Q\} &\equiv e \gg= (\lambda x \rightarrow \text{do } \{Q\}) \\ \text{do } \{e; Q\} &\equiv e \gg= (\lambda \_ \rightarrow \text{do } \{Q\}) \\ \text{do } \{e\} &\equiv e \end{aligned}$$

Thread soup	$P, Q$	$::= M_t \mid (P \mid Q)$
Descriptors	$D_\perp$	$::= M \cup \{\perp\}$
Heap	$\Theta$	$::= r \mapsto M \times D_\perp$
Allocations	$\Delta$	$::= r \mapsto M \times D_\perp$
Access types	$T$	$::= \{\text{CREATE, READ, WRITE}\}$
Log	$\Sigma$	$::= \text{list monoid } ([], \oplus) \text{ over } T \times D$
Evaluation contexts	$E$	$::= [\cdot] \mid E \gg= M \mid \text{catch } E M$
	$S$	$::= [\cdot] \mid S \gg= M$
	$\mathbb{P}$	$::= S_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P})$
Action	$a$	$::= !c \mid ?c \mid \epsilon$

I/O transitions $P; \Theta \xrightarrow{a} Q; \Theta'$		
$\mathbb{P}[\text{putChar } c]; \Theta$	$\xrightarrow{!c}$	$\mathbb{P}[\text{return } ()]; \Theta$ (PUTC)
$\mathbb{P}[\text{getChar } ]; \Theta$	$\xrightarrow{?c}$	$\mathbb{P}[\text{return } c]; \Theta$ (GETC)
$\mathbb{P}[\text{forkIO } M]; \Theta$	$\rightarrow$	$(\mathbb{P}[\text{return } t] \mid M_t); \Theta \quad t \notin \mathbb{P}, \Theta, M$ (FORK)
$\mathbb{P}[\text{catch (return } M) N]; \Theta$	$\rightarrow$	$\mathbb{P}[\text{return } M]; \Theta$ (CATCH1)
$\mathbb{P}[\text{catch (throw } M) N]; \Theta$	$\rightarrow$	$\mathbb{P}[N M]; \Theta$ (CATCH2)
$\frac{M \rightarrow N}{\mathbb{P}[M]; \Theta \rightarrow \mathbb{P}[N]; \Theta} \quad (\text{ADMIN})$		
$\frac{M; \Theta, \{\} \xrightarrow{*} \text{return } N; \Theta', \Delta'}{\mathbb{P}[\text{atomic } M]; \Theta \rightarrow \mathbb{P}[\text{return } N]; \Theta'}$		(ARET1)
$\frac{M; \Theta, \{\} \xrightarrow{*} \text{throw } N; \Theta', \Delta'}{\mathbb{P}[\text{atomic } M]; \Theta \rightarrow \mathbb{P}[\text{throw } N]; \Theta \cup \Delta'}$		(ATHROW1)
Administrative transitions $M \rightarrow N$		
$M$	$\rightarrow$	$V$ if $\mathcal{V}[M] = V$ and $M \neq V$ (EVAL)
$\text{return } N \gg= M$	$\rightarrow$	$M N$ (BIND)
$\text{throw } N \gg= M$	$\rightarrow$	$\text{throw } N$ (THROW)
$\text{retry } \gg= M$	$\rightarrow$	$\text{retry}$ (RETRY)
STM transitions $M; \Theta, \Delta, \Sigma \Rightarrow N; \Theta', \Delta', \Sigma'$		
$\mathbb{S}[\text{readTVar } r]; \Theta, \Delta, \Sigma$	$\Rightarrow$	$\mathbb{S}[\text{return } \Theta_1(r)]; \Theta, \Delta, \Sigma$ if $r \in \text{dom}(\Theta)$ and $\Theta_2(s) = \perp$ (READ)
$\mathbb{S}[\text{writeTVar } r M]; \Theta, \Delta, \Sigma$	$\Rightarrow$	$\mathbb{S}[\text{return } ()]; \Theta[r \mapsto (M, \perp)], \Delta, \Sigma$ if $r \in \text{dom}(\Theta)$ and $\Theta_2(s) = \perp$ (WRITE)
$\mathbb{S}[\text{newTVar } M]; \Theta, \Delta, \Sigma$	$\Rightarrow$	$\mathbb{S}[\text{return } r]; \Theta[r \mapsto (M, \perp)], \Delta[r \mapsto (M, \perp)], \Sigma$ $r \notin \text{dom}(\Theta)$ (NEW)
$\frac{M \rightarrow N}{\mathbb{S}[M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[N]; \Theta, \Delta, \Sigma} \quad (\text{AADMIN})$		
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{return } N; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{return } N]; \Theta', \Delta', \Sigma'}$		(OR1)
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{throw } N; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw } N]; \Theta', \Delta', \Sigma'}$		(OR2)
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{retry}; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[M_2]; \Theta, \Delta, \Sigma}$		(OR3)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{return } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[N M']; \Theta \cup \Delta', \Delta \cup \Delta', \Sigma \oplus (\Sigma' \uparrow_{\Delta'})}$		(XSTM2)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{return } M']; \Theta', \Delta \cup \Delta', \Sigma \oplus \Sigma'}$		(XSTM1)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{retry}]; \Theta, \Delta, \Sigma}$		(XSTM3)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{return } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{return } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{return } M']; \Theta', \Delta', \Sigma}$		(AURET1)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{return } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw } M']; \Theta', \Delta', \Sigma}$		(AUTHROW1)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{return } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{throw } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw UnauthorizedError}]; \Theta', \Delta', \Sigma}$		(AURET2)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{throw } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw UnauthorizedError}]; \Theta', \Delta', \Sigma}$		(AUTHROW1)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{retry}; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{retry}]; \Theta', \Delta', \Sigma}$		(AURETRY)

Figure 2: Operational semantics (part 1)

TMI transitions	
$\mathbb{S}[\text{readTMIVar } r]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{return } \Theta_1(r)]; \Theta, \Delta, \Sigma \oplus [(\text{READ}, \Theta_2(r))]$	$\text{if } r \in \text{dom}(\Theta) \text{ and } \Theta_2(r) \neq \perp$ (TMIREAD)
$\mathbb{S}[\text{writeTMIVar } r M]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{return } ()]; \Theta[r \mapsto (M, \Theta_2(r))], \Delta, \Sigma \oplus [(\text{WRITE}, \Theta_2(r))]$	$\text{if } r \in \text{dom}(\Theta) \text{ and } \Theta_2(r) \neq \perp$ (TMIWRITE)
$\mathbb{S}[\text{newTMIVar } N M]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{return } r]; \Theta[r \mapsto (M, N)], \Delta[r \mapsto (M, N)], \Sigma \oplus [(\text{CREATE}, N)]$	$r \notin \text{dom}(\Theta)$ (TMINEW)
$\frac{M \rightarrow N}{\mathbb{S}[M]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[N]; \Theta, \Delta, \Sigma}$	(TADMIN)
$\frac{M; \Theta, \Delta, \Sigma \xrightarrow{*} N; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{liftSTM } M]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[N]; \Theta', \Delta', \Sigma'}$	(LIFTSTM)
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{return } N; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{return } N]; \Theta', \Delta', \Sigma'}$	(TOR1)
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{throw } N; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{throw } N]; \Theta', \Delta', \Sigma'}$	(TOR2)
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{retry}; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[M_2]; \Theta, \Delta, \Sigma}$	(TOR3)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{return } M']; \Theta', \Delta \cup \Delta', \Sigma \oplus \Sigma'}$	(XTMI1)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[N M']; \Theta \cup \Delta', \Delta \cup \Delta', \Sigma \oplus (\Sigma'  _{\Delta'})}$	(XTMI2)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{retry}; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{retry}]; \Theta, \Delta, \Sigma}$	(XTMI3)
Authorization transitions	
$\frac{M \rightarrow N}{\Sigma \vdash \mathbb{E}[M] \rightsquigarrow \mathbb{E}[N]}$	(AUADMIN)
$\Sigma \vdash \mathbb{E}[\text{getlog}] \rightsquigarrow \mathbb{E}[\text{return } \text{hs}(\Sigma)]$	(GETLOG)
$\Sigma \vdash \mathbb{E}[\text{catch } (\text{return } M) N] \rightsquigarrow \mathbb{E}[\text{return } M]$	(ACATCH1)
$\Sigma \vdash \mathbb{E}[\text{catch } (\text{throw } M) N] \rightsquigarrow \mathbb{E}[N M]$	(ACATCH2)

Figure 3: Operational semantics (part 2)

Figure 4: Program state and evaluation contexts

Figure 4 defines some symbols used in the semantics. The metavariable  $D$  represents a set of terms used to describe the security properties of variables. We extend this set with an invalid value  $\perp$  and write  $D_\perp$  for the extended set. A state of a computation is a pair  $(M, \Theta)$  of a term that remains to be evaluated and a store  $\Theta$ . The store maps variable names to terms and their variable descriptor. If a variable does not have a suitable descriptor, we use  $\perp$  as a fill-in.

Access types  $T$  is a set of three constants, representing the operation performed on variables. An access log  $\Sigma$  is a list monoid of pairs  $(t, d)$  where  $t$  is an access type and  $d$  is a descriptor term, we use  $[]$  for the empty list and  $\oplus$  for concatenation, and in the semantics we use  $[\cdot]$  as a constructor. Other symbols are conventional and taken from [2].

For (partial) functions  $f$  whose co-domain is a cross-product of two or more sets, and an integer  $i$ ; we write  $f_i$  instead of  $\pi_i \circ f$  where  $\pi_i$  is the normal  $i$ -th projection function. For convenience, we introduce the following notation for filtering logs. If  $\Delta$  is a store and  $\Sigma$  is an access log, we define the  $\Delta$ -restriction of  $\Sigma$ , indicated by  $\Sigma|_\Delta$ , thus

$$\begin{aligned} []|_\Delta &= [] \\ ((t, d) \oplus \Sigma')|_\Delta &= \begin{cases} [(t, d)] \oplus \Sigma'|_\Delta & \text{if } d \in \text{img}(\Delta_2) \\ \Sigma'|_\Delta & \text{otherwise} \end{cases} \end{aligned}$$

Intuitively,  $\Sigma|_\Delta$  is the list of entries from  $\Sigma$  which apply to variables defined by  $\Delta$ , where variables are identified by their security descriptor.

## 4.2 Operational semantics

Figures 2 and 3 detail the transition relations of our semantics. The former is in principle the same as the semantics of [2]. The semantics use several different transition systems that are layered such that a series a reduction in one layer becomes one reduction in the next one above. Namely, there are three main layers - the top level I/O context, the STM context and the TMI context. An auxiliary transition system is used to reduce authorization functions.

The top level I/O actions are described by the  $\rightarrow$  relation. They operate on the  $\mathbb{P}$  context, which allows for picking any program term from the thread soup for reduction. The first two rules are I/O primitives. The rule *FORK* is used to create a new thread and enter it into the thread soup. The rules *CATCH1* and *CATCH2* deal with exception handling as described in the appendix of the post-publication, extended version of the Haskell Semantics [2]. The *ADMIN* rule allows for lifting of administrative transitions from a separate transition relation. This is done to reduce repetition, as the rules there also apply to the the STM and TMI transition relations. Finally, the rules *ARET1* and *ATHROW1* enable the use of the atomic combinator to lift a series of reductions in the STM transition relation to a single I/O transition.

If the series of STM reductions results in a **return** value, the effects on the store are retained. If it however results in an exception (i.e. a throw value), the modifications to existing variables are discarded but any new allocations are retained. This is necessary as the exception value may hold references to newly allocated variables.

The *admin* transitions define the evaluation of terms to values via a function  $\mathcal{V}$ . This function is standard and its definition omitted here. Administrative transitions also include the behaviour of the monadic bind operator  $\gg=$ .

The STM transitions define the behavior of STM actions. The states used in this transitions is extended from the I/O transitions by adding a separate store for new allocations  $\Delta$ , and an access log  $\Sigma$ . The first three rules define actions on transactional variables. They operate on the store  $\Theta$  but only on those variables where the second component (i.e. the security descriptor) is  $\perp$ . This is what differentiates regular transactional variables from security sensitive variables.

Other rules in the STM transition system define the behavior of STM combinators as described in section 2.2. We have used the revised semantics of exception handling from the later versions of [2], namely the rule *XTM2* ensure that when a term reduction results in an caught exception, all effects of that reduction are rolled back except for new allocations.

The most notable addition to the STM transitions is however the handling of *authorized*. The rules *AURETn* and *AUTHROWN* specify that for a term of the form *authorized N M*, if *M* evaluates to a **return** or **throw** value, that value is passed on only if *N* evaluates under the authorization relation (see later) to a **return** value. If *N*, the *authorization term*, evaluates to an exception an exception is raised. This results in the rollback of any updates that took place inside the TMI context, no matter if the exception is caught in the STM context or allowed to propagate up to the I/O context.

We should note that in the case of an exception, including authorization failure, new allocations are retained for the reason described above. In the implementation, this is not done explicitly as deallocation of references are handled by the garbage collector. Any new allocations that are actually referenced by exception values are thus retained, but others are discarded. Thus, since we don't allow any variable references in our special exception for authorization failures, no new allocations will leak in practice.

Figure 3 shows the two transition relations we have added. The first one deals with TMI actions. TMI actions behave very much like STM actions, The main difference is that variable operations in TMI actions can operate on security sensitive variables. These are the first three rules in figure 3. When these operations are performed, a log entry is added to  $\Sigma$  with the contents of the variable's security descriptor. Another addition over STM behavior is rule *LIFTSTM*. This rule states that any sequence of STM reductions can be lifted to the TMI level. This is necessary to allow TMI code to access regular transactional variables, and should be possible, since TMI actions are only performed in the context of an enclosing STM action.

Finally, we add a separate transition system to evaluate authorization functions. This transition system only allows pure operations and monad binding via the administrative transitions, the usual exception handling and one special term *getlog*. The authorization transitions take place in the context of a log  $\Sigma$ , the *getlog* converts this log to a Haskell list which user supplied code can then inspect.

## 5 Implementation

Our Haskell implementation is comprised of one module, *TMI*. Most important are the monad *TMI d a* and the type for TMI variables, *TMIVar d a*. Both are parameterized on the descriptor type *d*, which is chosen by the user of this module. A TMI variable is represented by a descriptor value and an STM *TVar*,

```
data TMIVar d a = TMIVar {
  getTVar      :: TVar a,
  getDescriptor :: d
}
```

The field accessors *getTVar* and *getDescriptor* are not exported and only available inside the *TMI* module.

The *TMI* monad is a stack of the well known *writer monad* on top of the regular *STM* monad. In the *TMI* module, the regular *STM*

module is imported under the name *T*.

```
newtype TMI d a = TM {
  unwrapTM :: WriterT (InternalTMILog d) T.STM a
} deriving (Monad, MonadWriter (InternalTMILog d))
```

The type *InternalTMILog d* represents a log of all accesses to TMI variables. It is defined by the following declarations.

```
type Elevation = String
```

```
data TMILogEntry d
  = TMICreate d
  | TMIRead d
  | TMIWrite d
  | TMIElevate Elevation
  | TMIPopElevation
```

```
type InternalTMILog d = [TMILogEntry d]
```

An entry of the form *TMIRead x* just means that a variable with descriptor value *x* was read. The log can also contain elements that signify the start or end of a sequence of *elevated* operations, which we have not discussed before. This is needed, for example, in code that calculates aggregates of values that the current user does not have access to. The aggregating code pushes a *TMIElevate e* where *e* is some application specific identifier. It then performs the operations needed to calculate the aggregate value and then signifies end of the elevated section by adding *TMIPopElevation* to the log. As such, these markers always come in pairs. This is ensured by the following function, which provides the only access to elevations to other modules.

```
elevated :: Elevation -> TMI d a -> TMI d a
elevated level action =
  do tell [TMIElevate level]
     rval <- action
     tell [TMIPopElevation]
  return rval
```

The *liftSTM* function lifts an STM operation to a TMI operation. The log is not affected by the work performed in the STM action.

```
liftSTM :: STM a -> TMI d a
liftSTM = TM . lift
```

The functions to create, read and write TMI variables are now simple to define. They all enter the relevant entries to the log and then call the underlying functions from the *STM* module.

```
newTMIVar :: d -> a -> TMI d (TMIVar d a)
newTMIVar description val =
  do tell [TMICreate description]
     var <- liftSTM $ T.newTVar val
  return $ TMIVar var description
```

```
readTMIVar :: TMIVar d a -> TMI d a
readTMIVar tv =
  do tell [TMIRead $ getDescriptor tv]
     liftSTM $ T.readTVar $ getTVar tv
```

```
writeTMIVar :: TMIVar d a -> a -> TMI d ()
writeTMIVar tv val =
  do tell [TMIWrite $ getDescriptor tv]
     liftSTM $ T.writeTVar (getTVar tv) val
```

The combinators from the *STM* world are defined thus.

```
retryTMI = liftSTM T.retry
```

```

orElseTMI t1 t2 = TM . WriterT $ runTMI t1 `T.orElse`
  runTMI t2

```

What is left is to define the crucial authorized function. This function accepts an authorization function with the type `InternalTMILog d -> Bool` and a TMI action; and it should return an STM action that performs the operation of the TMI action, validates the resulting log with the authorization function and either returns the result or throws an exception. With this description in mind, the implementation is pretty straight-forward.

However, we do not want the user to worry about the pairs of elevation markers, and as the name `InternalTMILog d` implies, that is not what we will expose to the authorization function. Instead, we define types that are generally simpler to handle in authorization functions.

```

data TMIAccess = CreateVar | ReadVar | WriteVar
type TMILog d = [(TMIAccess, d, Maybe Elevation)]

```

The idea is that entries that the authorization function needs to validate are triples of an access type specifier, the descriptor of the variable and the elevation level that was active at the time of access, if any. The `expandLog` function does the leg-work of converting between our internal log representation and this one.

```

expandLog :: InternalTMILog d -> TMILog d
expandLog = expand' [] [Nothing]
  where
    expand' c stack ((TMIElevate newlevel):rest) =
      expand' c (Just newlevel:stack) rest

    expand' c (_:stack) (TMIPopElevation:rest) =
      expand' c stack rest

    expand' c stack@(level:_) ((TMICreate d):rest) =
      expand' ((CreateVar, d, level):c) stack rest

    expand' c stack@(level:_) ((TMIRead d):rest) =
      expand' ((ReadVar, d, level):c) stack rest

    expand' c stack@(level:_) ((TMIWrite d):rest) =
      expand' ((WriteVar, d, level):c) stack rest

    expand' c _ [] = c

```

Now, the implementation of `authorized` turns out to be relatively straight-forward.

```

authorized :: (TMILog d -> Bool) -> TMI d a -> T.STM a
authorized auth act = do
  (result, log) <- runTMI act
  if not . auth $ expandLog log
  then throw $ AssertionFailed "Access_denied"
  else return result

```

## 6 Conclusion and future work

We have presented both a formal semantics and an implementation of TMI over the Haskell STM system. During this work, we discovered that there are many design decisions to be made and the design we have presented here is only one of many possibilities. Specifying a design formally with operational semantics is essential to evaluate design variants and identify their pros and cons.

One thing the design presented here does not provide is support for stateful policies. As discussed in our previous paper [1] we believe that there are no major obstacles to supporting this. The policy state must

be protected by the STM mechanisms to avoid race conditions or interference between threads, but we must take care that the accesses to this state are only performed by trusted code, as they cannot be subject to authorization, lest we introduce a circular dependency.

In our implementation we are maintaining the access log by hand. This works well for prototypical purposes, but we would like to investigate the possibility of making use of the real underlying transaction log. This requires modifications to the STM framework provided in the GHC library.

Future work in this context also involves writing or porting complex software to the architecture to obtain realistic performance measurements.

Most importantly, we think that having clear semantics for TMI and an implementation over a production-ready STM system, further validates our claim that TMI architecture is very relevant to practical software development.

## References

- [1] A. Birgisson, M. Dhawan, Ú. Erlingsson, V. Ganapathy, and L. Iftode, *Security enforcement using software transactional memory*, ACM Conference on Computer and Communications Security, October 2008.
- [2] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, *Composable memory transactions*, ACM Conference on Principles and Practice of Parallel Programming, 2005.
- [3] M. Herlihy, V. Luchango, and M. Moir, *A flexible framework for implementing software transactional memory*, ACM SIGPLAN OOPSLA, Oct 2006.
- [4] S. Peyton Jones, *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell*, Engineering theories of software construction, Marktoberdorf Summer School 2000 (M. Broy C. Hoare and R. Steinbrueggen, eds.), NATO ASI Series, IOS Press, 2001, pp. 47–96.
- [5] S. Peyton Jones and P. Wadler, *Imperative functional programming*, 20th ACM Symposium on Principles of Programming Languages (POPL'93), 1993, pp. 71–84.
- [6] Gordon D. Plotkin, *A structural approach to operational semantics*, Journal of Logic and Algebraic Programming 60-61 (2004), 17–139.
- [7] A. Gordon S. Peyton Jones and S. Finne, *Concurrent haskell*, 23rd ACM Symposium on Principles of Programming Languages (POPL'96), 1996, pp. 295–308.