

An Implementation and Semantics for Transactional Memory Introspection in Haskell

Arnar Birgisson
Reykjavík University
arnarb07@ru.is

Úlfar Erlingsson
Reykjavík University
Microsoft Research, Silicon Valley
ulfar@ru.is

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. TMI builds on and integrates with implementations of the Software Transactional Memory (STM) architecture [Harris and Fraser 2003]. In this paper we present a formal definition of TMI and a concrete implementation over the Haskell STM. We find that this specification and reference implementation establishes clear semantics for the TMI architecture. In particular, they help identify and resolve ambiguities that apply to implementations such in our prior work [Birgisson et al. 2008].

Categories and Subject Descriptors D.4.6 [Operating Systems]: Access controls; D.1.3 [Software]: Concurrent programming

General Terms Languages, Security

Keywords Reference monitors, Transactional memory

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 [Birgisson et al. 2008] we presented Transactional Memory Introspection (TMI), an architecture that greatly simplifies the implementation of correct reference monitors on mechanisms that implement Software Transactional Memory (STM) [Harris and Fraser 2003, Herlihy and Moss 1993] support. In this paper we present a formal semantics for TMI, as well as a reference TMI imple-

mentation over the Haskell STM. These specifications clarify the TMI architecture and help identify and resolve ambiguities in its implementation.

STM systems provide many useful guarantees that make the implementation of multithreaded software easier and less error-prone. In particular, STM offers atomicity and isolation through optimistically executing concurrent code and monitoring for conflicting accesses to resources. By providing rollback mechanisms, STM systems can resolve conflicting accesses by undoing the work of a transaction and retrying that transaction again, from the beginning.

All STM implementations must perform bookkeeping of accesses (such as reads and writes) to shared resources. By imposing on this bookkeeping, and the necessary monitoring and validation steps, TMI provides facilities to support the creation of robust and correct enforcement mechanisms. TMI provides *complete mediation* by enhancing the STM runtime checks against conflicting, concurrent accesses, and TMI adds the requirement that all accesses must have been successfully authorized before a transaction is committed. TMI also *simplifies error handling*. When unauthorized accesses are detected in a transaction, the transaction is rolled back and not retried. This saves the programmer from the onerous and error-prone task of performing clean-up after a failed authorization.

Another common problem in traditional authorization is *time of check to time of use* bugs. Such bugs arise when an authorization check is used to decide if a dangerous operation should be performed, and when the interleaving of code execution may cause state changes that invalidate that decision, before the dangerous operation is actually performed. TMI resolves this problem, by making use of STM mechanisms to execute both the authorization check and the dangerous operation within a single transaction.

In [Birgisson et al. 2008] we give a comprehensive, informal overview of the TMI architecture and also evaluate a Java TMI implementation built on the DSTMM2 library [Herlihy et al. 2006]. In this previous, companion paper, we also discuss the relationship between TMI and other approaches,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLAS '09 June 15, Dublin, Ireland.

Copyright © 2009 ACM 978-1-60558-645-8/09/06...\$10.00

such as aspect-oriented and transactional techniques for security enforcement.

In this current paper, we give a more formal treatment of TMI, and provide a clear, well-defined structural operational semantics [Plotkin 2004] for the TMI architecture. Our TMI semantics builds on the well founded semantics for the the Haskell STM system in [Harris et al. 2005], and is accompanied by an implementation over the Haskell STM system.

We found that the development of a formal semantics alongside an implementation helped us disambiguate design choices and resolve ambiguities. In particular, the formal semantics allowed us to safely combine multiple TMI actions and different security managers into a single, atomic authorization decision. The implications of such compositionality are not clear, given only informal reasoning, and, indeed, some of our initial implementation strategies did not provide correct enforcement. However, as described further in Section 4, when combined with a formal semantics, we can establish that our TMI implementation correctly enforces the intended security policy.

The structure of the paper is as follows. In Section 2 we give the necessary background, including STM systems and how TMI makes use of their mechanisms, as well as an overview of the Haskell STM implementation. Section 3 covers TMI in greater detail and describes its Haskell implementation from the user standpoint. Section 4 defines the formal semantics of TMI, building on existing semantics for STM Haskell. Section 5 describes the key elements of our Haskell implementation and Section 6 discusses future work.

2. Background

2.1 STM and TMI

STM provides attractive guarantees for multithreaded software; namely atomicity, consistency and isolation of specifically marked blocks of code in *transactions*. In general, STM implementations must do so by performing

- careful monitoring of the resources that are accessed within a transaction,
- validation of the accesses 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 [Birgisson et al. 2008] 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 validation checks are able to inspect the resource access logs of the STM and may veto the commit if an application specific policy is violated. In general, this requires that STM mechanisms provide 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 isolated and cannot affect the policy decisions of 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 raise 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 [Weimer and Necula 2008].

2.2 Haskell STM

For a formal treatment, we build our semantics and implementation on those of the Haskell STM [Harris et al. 2005], which in turn is built on Concurrent Haskell [Peyton Jones et al. 1996]. 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. Non-pure computations are modelled with *monads* [Peyton Jones and Wadler 1993]; this includes computations with side-effects such as input/output and mutable state.

The main entry to a Haskell program is an instantiation of the I/O monad, i.e. a value that represents an *action* of the type `IO ()`. An action of this type can, in addition to performing pure computation, perform other I/O actions by way of composing smaller actions into larger ones. For an example, Haskell standard libraries define the basic I/O actions `getChar` and `putChar`, which read from standard input and write to standard output, respectively. The most common composition is simple sequencing. For example the composed I/O action

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

defines an action that, when executed, will perform the three actions listed in sequence.

In general a value of type `IO a` represents an action that when executed, may perform some I/O operations as defined by the Haskell libraries and then result in a value of type `a`. Pure functions cannot execute such actions without jeopardizing their purity and this is neatly enforced by the Haskell type system. Naturally, I/O actions are however free

to run pure computations. Thus the only way to get at the value of an action is if it is a part of a bigger I/O action. The Haskell runtime bootstraps the whole process by executing the special I/O action called `main`.

In addition to conventional input and output, I/O actions can perform reads and updates of mutable memory cells. The type `IORef a` represents a mutable cell that contains a value of type `a`. Haskell provides the basic I/O actions `newIORef`, `readIORef` and `writeIORef` for manipulation of such cells. As with other I/O actions, these operations can only be used when composing larger I/O actions.

Concurrent Haskell supports explicit forking of threads through the I/O action `forkIO`.

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

`forkIO` takes another 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 [Peyton Jones 2001] or tutorials such as [Peyton Jones and Singh 2008].

The Haskell STM is based on a monadic type similar to the one for I/O actions, namely `STM a`. A value of this type represents an *STM action*, which when executed may perform smaller STM actions and result in a value of type `a`. STM actions may contain pure computations as well, but note that they *cannot* contain e.g. I/O actions. The main STM actions provided are actions that allow manipulations of another kind of memory cells, which have the type `TVar a`, where `a` is the type of value that the cell holds. The actions are `newTVar`, `readTVar` and `writeTVar`, so they have the same power as their I/O counterparts. The important thing to note is that sets of `IORefs` and `TVars` are kept separate; one can only be used in I/O actions and the other in STM actions.

STM actions can be composed. Similar to I/O actions, the most common composition is sequencing, but in addition Haskell STM provides the basic STM action `retry` and a combinator `orElse`. The action `retry` is a blocking operation for STM actions, which restarts the current transaction with potentially updated `TVar` contents. By issuing `retry`, the programmer is stating that the current transaction cannot finish for the state of `TVars` it started in. The Haskell STM provides an optimized implementation of `retry`. This implementation captures the set of `TVars` that a transaction has read before the `retry`, and suspends the transaction until at least one of those `TVars` has been updated. This makes sense because the *only* outside factors that can affect the execution of an STM action are the values of the `TVars` it reads.

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` invokes the `retry` action, then the combined action rolls back the effects of `t1` and tries `t2` instead. If that one retries also, the whole action retries, 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 [Harris et al. 2005].

While the basic STM actions and their compositions give us a way to build larger STM actions, we have not discussed how those actions can be run or how they relate to transactions. For this, STM Haskell provides us with the `atomically` function,¹ whose type is

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

This function gives an I/O action that, when performed, will execute the input STM action. The atomicity comes from the fact that STM Haskell will guarantee that the effects that the STM action has on `TVars` are atomic, i.e. they all become visible at once and that what happens inside the STM action is not affected by concurrent threads.

The Haskell STM system does this by monitoring concurrent invocations of STM actions, taking care of rolling them back if they conflict with each other and retrying them. 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))
```

The `increment` action is a classical example of where a race condition might occur in a traditional setting, but in our situation the STM system will guarantee the atomicity of each invocation.

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 work [Birgisson et al. 2008], 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 can be used to eliminate concerns about check placement, race conditions and exceptional execution paths.

TMI provides these guarantees by imposing on the STM system. The programmer marks certain variables as security

¹ While [Harris et al. 2005] uses the name `atomic`, the actual implementation of the Glasgow Haskell Compiler uses `atomically`.

sensitive. This implicitly indicates to the STM system that these variables are shared, and ensures that the STM system will protect against race conditions in accesses to the variables. TMI enhances the monitoring of these security sensitive variables by ensuring that an access-control reference monitor is invoked every time that the variables are accessed.

Time of policy evaluation with TMI: The TMI architecture only loosely constrains when a policy must be evaluated, and in [Birgisson et al. 2008] we consider a number of alternatives. In particular, TMI enforcement can be *eager* or *lazy*. With *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. With *lazy* enforcement, accesses to variables are simply logged (often they are already logged by the STM) and the logs are inspected by the reference monitor only at the end of the transaction. If any of the logged accesses are invalid, the whole transaction is aborted.

A key property of TMI enforcement is that policy decisions can be evaluated at any time, as long as they are evaluated in a serialized fashion, and evaluation is fully complete before the transaction commits. A good STM system will ensure that each transaction is executed in isolation, such that aborting one will have the same semantics as not having started it. This said, 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.

Utilizing TMI enforcement: To use TMI, the programmer declares a set of variables as security relevant. This implicitly indicates to the underlying STM system that those variables should be protected against race conditions. This means the values held by these variables can only be read or modified within a transaction, and that the STM system takes care of resolving conflicting accesses by concurrent transactions. This also means that, upon every variable access, TMI appends information identifying the variable in question to a transaction-specific *introspection log*. In particular, the introspection log will contain information about the creation, reading, and writing of the security sensitive variables.

In addition, TMI requires that all sections of code that access security-sensitive variables must be explicitly marked as *atomic*. To execute such atomic code sections, programmers initiate a TMI transaction and provide a reference to the atomic block and a *security manager*. The security manager is a block of code (or closure) that encodes the intended, application-specific security policy, and is able to determine whether a transaction introspection log satisfies the security policy. The security manager closure includes the active principal, and other auxiliary information that is needed to check policy compliance.

Finally, transaction commit plays a special role in TMI enforcement. TMI runs atomic blocks as transactions in the

underlying STM system, but changes the semantics of transaction commit. After a TMI atomic block has finished execution, but before it is committed, TMI ensures that the security manager has fully evaluated whether the transaction introspection log complies with the intended security policy. Importantly, this evaluation occurs within the same STM transaction as the execution of the atomic block, and a commit of the transaction is attempted only if the security manager returns success.

Even when the transaction has complied with the security policies, the attempted commit may still fail, and the transaction is retried, in the case when the STM system finds conflicting concurrent accesses. Also, if the security manager finds the transaction in violation of policy, all state changes are rolled back—including changes to the security manager state, in the case of history-based policies—and, instead of retrying the transaction, an exception is raised to the invoker of the atomic block.

A simple example: The following pseudo code shows what software that makes use of TMI-based security enforcement might look like. (Note that the code makes use of function-argument currying.)

```
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
```

In this code, two aspects are especially noteworthy. First, security enforcement code is completely decoupled from the application logic and the function `withdraw` performs no authorization. Even so, complete mediation is ensured, since the introspection log is implicitly updated by the TMI reference monitor upon each access to account variables.

Second, in the case of authorization failure (e.g. a withdrawal from a different user's account), the error handler need only consider how to indicate the error to the user. The error handler need not clean up any mess: the state changes that happened during the transaction (if any) have already been rolled back when the error handler starts execution. Although perhaps not apparent in this simplified example, there is ample evidence that writing correct cleanup code is diffi-

cult, especially when multiple security-relevant operations are involved [Weimer and Necula 2008].

While the above observations form the two main benefits of the TMI architecture, the third is freedom from TOCTTOU bugs. Without TMI, this example code might suffer from TOCTTOU race conditions, e.g., if accounts could change owners. However, with TMI, such account-ownership changes would be isolated, and a transaction would be guaranteed to see the same owner throughout its execution.

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 provides 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.

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 an existing Haskell STM program can be easily adapted to TMI with minimal changes to their 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. Security descriptors are provided by the programmer when she creates sensitive variables and contain the metadata about the variable that is necessary for authorization, such as the owner of an account, permissions of a file, etc.

Since the type of security descriptors is application 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 ()
```

```
x, y ∈ Variable
r, t ∈ Name
c ∈ Char

V ::= r | c | \x->M
    | return M | M >>= N
    | putChar c | getChar
    | throw M | catch M N
    | retry | M `orElse` N
    | forkIO M | atomically M
    | newTVar M
    | readTVar r | writeTVar r M
    | newTMIVar N M
    | readTMIVar r | writeTMIVar r M
    | authorized N M | liftSTM M
    | getlog | UnauthorizedError

M, N ::= x | V | M N | ...
```

Figure 1. Syntax of values (*V*) and terms (*N, M*)

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

```
-- Security descriptor for accounts
data AccountDescr = AccountDescr {
    acctOwner :: String,
    acctNumber :: Int
}
type Account = TMIVar AccountDescr Int

createAccount :: String -> Int -> Int
                -> TMI Account
createAccount owner number balance =
    newTMIVar (AccountDescr owner number) balance
```

The next function demonstrates reading and writing of the security-relevant account variables.

```
deposit :: Account -> Int -> TMI AccountDescr ()
deposit acct amount =
    do balance <- readTMIVar acct
       writeTMIVar 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 transaction introspection log of security-relevant accesses and determines if the transaction should be aborted. As an input to this function, TMI defines the type of an introspection 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, and this currying ensures that we satisfy the type required by authorized.

```
auth :: String -> TMILog AccountDescr -> Bool
auth user thelog = all checkowner thelog
  where
    checkowner :: (AccessType, AccountDescr)
                -> Bool
    checkowner (_,descr) =
      user == (acctOwner descr)

-- Defined by the TMI module:
-- authorized :: (TMILog d -> Bool)
--             -> TMI d a
--             -> STM a

main =
  do acct <- atomically (allowAll mkAccount)
     atomically (doDeposit acct "alice") -- OK
     atomically (doDeposit acct "bob")  -- FAILS
  where
    mkAccount = createAccount "alice" 123456 0
    doDeposit acct user =
      authorized (auth user) (deposit acct 42)
```

Since TMI actions are ultimately executed as 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 `authorized`. While this might cause ambiguity for other implementations, in this Haskell-based implementation such nesting has clear and well-defined semantics, and can therefore be permitted. In fact, we will make explicit use of such nesting in the following sections to implement *privilege amplification*.

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 `TMIVar`-actions that are actually committed or could have affected the committed actions.

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

Figure 2. Program state and evaluation contexts

4. Formal semantics of TMI

To formalize the semantics of TMI, we build on the semantics for the Haskell STM presented in [Harris et al. 2005]. The semantics is a structural operational semantics in the style of Plotkin [Plotkin 2004]. For the sake of completeness and to help the reader understand our extensions, we give a cursory explanation of the concepts of the semantics from [Harris et al. 2005] so that a reader not familiar with it may understand our extensions.

It is not obvious that the ideas presented in the previous section are indeed always safe. For example, we could not be sure that the nesting of TMI actions inside an STM action — in turn lifted to yet another TMI action — would result in a reasonable behavior. The construction of the following semantics greatly clarified our understanding of such subtleties. Our first drafts of the semantics revealed several ambiguities that were later resolved. Furthermore, constructing semantics for our intermediate implementation ideas sometimes revealed cases where incorrect behavior was possible, and the intended security guarantees of TMI were violated. Initially, for instance, a clear distinction of security-relevant data was missing and, while TMI actions naturally supported STM functionality, we had to explore several options to find the support for flexible STM and TMI combinations that is present in the final semantics.

In particular, the final semantics provides a clear separation between TMI and STM actions, which allows STM actions to be lifted to the TMI level and ensures correct behavior when nesting TMI and STM actions, or several TMI actions, one within another. This nesting support provides powerful composability properties, and makes it possible to safely combine multiple TMI actions and different security managers into a single, atomic authorization decision. Without a formal semantics, the implications of such composability would have been unclear, and its correctness suspect. However, in our final semantics, given below, it is straightforward to see that the TMI security policy enforcement guarantees are always correctly maintained.

Figures 3 through 5 give the operational rules that describe the steps a program may take. At the top level, a pro-

Administrative transitions	$M \rightarrow N$
----------------------------	-------------------

$$\begin{array}{l}
M \rightarrow V \text{ if } \mathcal{V}[M] = V \text{ and } M \neq V \quad (EVAL) \\
\text{return } N \gg= M \rightarrow M N \quad (BIND) \\
\text{throw } N \gg= M \rightarrow \text{throw } N \quad (THROW) \\
\text{retry } \gg= M \rightarrow \text{retry} \quad (RETRY)
\end{array}$$

Figure 3. Evaluation of terms and monad operations

gram transforms a state of the form $P; \Theta$ via a labelled transition.

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

P represents a program term in the syntax of Figure 1 while Θ 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 program's input and output actions. Q and Θ' 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 [Harris et al. 2005] there is no need to represent rollback, but contrary to the semantics in that paper, 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 standard except that the application of some monadic operators are considered values, a technique again lifted from [Harris et al. 2005]. The `do`-notation used up until now is standard syntactic sugar for the monad bind and return operations.

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

Figure 2 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, Θ) of a term that remains to be evaluated and a store Θ . The store maps variable names to terms and their variable descriptors. If a variable does not have a suitable descriptor, we use \perp as a fill-in. This is used to distinguish security-relevant variables from other variables.

The set of access types, T , consists of three constants, each representing an operation performed on variables. An introspection log Σ 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 [Harris et al. 2005].

For a (partial) function 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 π_i is the standard i -th projection function. For convenience, we introduce the following notation for filtering logs. If Δ is a store and Σ is an introspection log, we define the Δ -restriction of Σ , 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 Σ which apply to variables defined by Δ , where variables are identified by their security descriptors.

Interleaving of operations is modelled with the evaluation context \mathbb{P} , often referred to as a *thread soup*. Through this evaluation context the semantics can non-deterministically choose a term for reduction from the parallel construct, each term representing a thread. Haskell terms are usually reduced according to the evaluation context \mathbb{E} , which allows for reductions of the right hand side of the $\gg=$ operator as well as within the body of a `catch` term. However, since we want to handle exceptions in a specific manner for STM and TMI actions, we will use the simpler context \mathbb{S} which requires the operational semantics rules to specify explicitly how `catch` terms are handled.

4.2 Operational semantics

Figures 3 through 5 detail the transition relations of our semantics. Figures 3 and 4 are mostly the same as in the semantics of [Harris et al. 2005], parts added for TMI are indicated with a darker ink. The semantics uses several different transition systems that are layered such that a sequence of reductions in one layer becomes one reduction in the next layer above. This makes a sequence of transitions in a lower layer appear as one atomic transition at the higher level. There are three main layers - the top level I/O context, the STM context and the TMI context. Although the STM and TMI contexts

I/O transitions $P; \Theta \xrightarrow{a} Q; \Theta'$		
$\mathbb{P}[\text{putChar } c]; \Theta$	$\xrightarrow{!c}$	$\mathbb{P}[\text{return } ()]; \Theta$ <i>(PUTC)</i>
$\mathbb{P}[\text{getChar}]; \Theta$	$\xrightarrow{?c}$	$\mathbb{P}[\text{return } c]; \Theta$ <i>(GETC)</i>
$\mathbb{P}[\text{forkIO } M]; \Theta$	\rightarrow	$(\mathbb{P}[\text{return } t] \mid M_t); \Theta \quad t \notin \mathbb{P}, \Theta, M$ <i>(FORK)</i>
$\mathbb{P}[\text{catch } (\text{return } M) N]; \Theta$	\rightarrow	$\mathbb{P}[\text{return } M]; \Theta$ <i>(CATCH1)</i>
$\mathbb{P}[\text{catch } (\text{throw } M) N]; \Theta$	\rightarrow	$\mathbb{P}[N M]; \Theta$ <i>(CATCH2)</i>
$\frac{M \rightarrow N}{\mathbb{P}[M]; \Theta \rightarrow \mathbb{P}[N]; \Theta} \quad (\text{ADMIN})$		
$\frac{M; \Theta, \{\} \xrightarrow{*} \text{return } N; \Theta', \Delta'}{\mathbb{P}[\text{atomically } M]; \Theta \rightarrow \mathbb{P}[\text{return } N]; \Theta'} \quad (\text{ARET}) \quad \frac{M; \Theta, \{\} \xrightarrow{*} \text{throw } N; \Theta', \Delta'}{\mathbb{P}[\text{atomically } M]; \Theta \rightarrow \mathbb{P}[\text{throw } N]; \Theta \cup \Delta'} \quad (\text{ATHROW})$		

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 $\Theta_2(s) = \perp$ <i>(READ)</i>
$\mathbb{S}[\text{writeTVar } r M]; \Theta, \Delta, \Sigma$	\Rightarrow	$\mathbb{S}[\text{return } ()]; \Theta[r \mapsto (M, \perp)], \Delta, \Sigma$ if $r \in \text{dom}(\Theta)$ and $\Theta_2(s) = \perp$ <i>(WRITE)</i>
$\mathbb{S}[\text{newTVar } M]; \Theta, \Delta, \Sigma$	\Rightarrow	$\mathbb{S}[\text{return } r]; \Theta[r \mapsto (M, \perp)], \Delta[r \mapsto (M, \perp)], \Sigma$ $r \notin \text{dom}(\Theta)$ <i>(NEW)</i>
$\frac{M \rightarrow N}{\mathbb{S}[M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[N]; \Theta, \Delta, \Sigma} \quad (\text{AADMIN})$		
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{return } N; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{return } N]; \Theta', \Delta', \Sigma'} \quad (\text{OR1})$		
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{throw } N; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw } N]; \Theta', \Delta', \Sigma'} \quad (\text{OR2})$		
$\frac{M_1; \Theta, \Delta, \Sigma \xrightarrow{*} \text{retry}; \Theta', \Delta', \Sigma'}{\mathbb{S}[M_1 \text{ 'orElse' } M_2]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[M_2]; \Theta, \Delta, \Sigma} \quad (\text{OR3})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{return } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{return } M']; \Theta', \Delta \cup \Delta', \Sigma \oplus \Sigma'} \quad (\text{XSTM1})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[N M']; \Theta \cup \Delta', \Delta \cup \Delta', \Sigma \oplus (\Sigma' _{\Delta'})} \quad (\text{XSTM2})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{retry}; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{retry}]; \Theta, \Delta, \Sigma} \quad (\text{XSTM3})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{return } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{return } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{return } M']; \Theta', \Delta', \Sigma} \quad (\text{AURET1})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{return } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw } M']; \Theta', \Delta', \Sigma} \quad (\text{AUTHROW1})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{return } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{throw } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw UnauthorizedError}]; \Theta', \Delta', \Sigma} \quad (\text{AURET2})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{throw } M'; \Theta', \Delta', \Sigma' \quad \Sigma' \vdash N \xrightarrow{*} \text{throw } N'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{throw UnauthorizedError}]; \Theta', \Delta', \Sigma} \quad (\text{AUTHROW1})$		
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{retry}; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{authorized } N M]; \Theta, \Delta, \Sigma \Rightarrow \mathbb{S}[\text{retry}]; \Theta', \Delta', \Sigma} \quad (\text{AURETRY})$		

Figure 4. Operational semantics for IO and STM actions

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)
$\frac{M; \Theta, \{\}, [] \xrightarrow{*} \text{return } M'; \Theta', \Delta', \Sigma'}{\mathbb{S}[\text{catch } M N]; \Theta, \Delta, \Sigma \rightarrow \mathbb{S}[\text{return } M']; \Theta', \Delta \cup \Delta', \Sigma \oplus \Sigma'}$	(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 5. Operational semantics for TMI actions

are similar, a clean separation is crucial for a clean, yet powerful, specification of the semantics. An auxiliary transition system is used to reduce authorization functions.

Values and I/O transitions: The *admin* transitions of Figure 3 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 top level I/O actions are described by the labelled \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, choosing a fresh thread id t . The rules *CATCH1* and *CATCH2* deal with exception handling as described in the

appendix of the post-publication, extended version of the Haskell semantics [Harris et al. 2005]. The *ADMIN* rule allows for lifting of administrative transitions to the I/O transition relation. This is done to reduce repetition, as the administrative rules also apply to the STM and TMI transition relations, which have a similar lifting rule. Finally, the rules *ARET* and *ATHROW* enable the use of the atomic combinator to lift a sequence 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.

STM transitions: The STM transitions define the behaviour of STM actions. The states used in these transitions are extended from the I/O transitions by adding a separate store for new allocations Δ , and an introspection log Σ . A transition of the form

$$M; \Theta, \Delta, \Sigma \Rightarrow N; \Theta', \Delta', \Sigma'$$

represents a reduction of the term M to the term N . Some variables in Θ may be introduced or altered to yield Θ' . Δ is a store similar to Θ , that only tracks newly allocated variables while Σ is a log of accesses to TMI variables. Δ and Σ are transaction local, i.e. they are reset at the start of each atomic sequence of reductions in the STM system, see e.g. rule (*ARET*).

The first three rules define actions on transactional variables. They operate on the store Θ 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 [Harris et al. 2005], namely the rule *XTM2*, to ensure that when a term reduction results in a caught exception, all effects of that reduction are rolled back except for new allocations.

TMI transitions: A key TMI addition to the STM transitions is the handling of *authorized*. The rules *AU-RET_n* and *AUTHROW_n* specify that for a term of the form (*authorized N M*) if M evaluates to a return or throw value, then that value is propagated only if N evaluates to a return value under the authorization relation (see later). If evaluation of the *authorization term* N raises an exception,² a fixed exception containing no information about the local transaction state is thrown. This triggers a rollback of any updates performed during that invocation of *authorized*.

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 is 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 5 shows the two new TMI transition relations. The first one deals with TMI actions and is indicated by the symbol \rightarrow . The configurations of this transition system are identical to those of the STM system. Indeed, TMI actions behave very much like STM actions, the main difference is that variable operations in TMI actions can operate on

security sensitive variables. A variable v is security sensitive if and only if $\Theta_2(v) \neq \perp$. The first three rules of Figure 5 describe the variable operations. When these operations are performed, a log entry is added to Σ 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 always performed in the context of an enclosing STM action.

Finally, we add a separate transition system to evaluate authorization functions. In this system a transition of the form

$$\Sigma \vdash M \rightsquigarrow N$$

represents the reduction of term M to term N , under the context of an introspection log Σ . The reason for this notation is that the introspection log is fixed, i.e. read-only, for these transitions. This system only allows pure operations and monad binding via the administrative transitions, the usual exception handling and one special term *getLog*. The *getLog* term is reduced to a list representation (in the Haskell sense) of the access log Σ . The terms reduced with this system can thus examine the log and make decisions based on its contents.

An example: As an example of reading and applying the rules, consider the program

```
atomically
(
  authorized (assert (isEmpty getlog))
             (writeTMIVar x 10)
)
```

Working from the inside out, we can see that the innermost expression of *writeTMIVar x 10* will update the value of x in Θ as well as enter an entry to the introspection log Σ , by applying rule (*TMIWRITE*). The resulting term is *return ()*. As the resulting log is non-empty, the authorization term *assert (isEmpty getlog)* will throw an exception. Thus, for the *authorized* term, rule (*AURET2*) is the only applicable one, so that term evaluates to *throw UnauthorizedError*. The *atomically* term is therefore evaluated to the same result via rule (*ATHROW*), but this rule does not preserve updates to the store Θ , meaning that the transaction has been aborted.

Nested TMI actions: As we mentioned in the previous section, the capability of lifting STM actions up to the TMI levels allows us to nest TMI actions. An inner TMI action can be authorized with a separate authorization term. Consider the following example of an action that provides a student with information about her grade for a course, as well as the average of all grades of other students. Naturally, the student doesn't have access to other students' grades but for the purpose of calculating the average we may allow such access in a nested action.

²In the actual implementation the authorization term is simply a boolean function of the log. This difference exists to simplify both the semantics and the implementation.

Assume that we have defined the following terms.

- `ownGradesRead s` is an authorization term that succeeds only if the input log only contains reading of grades that belong to student `s`
- `allGradesRead` is an authorization term that succeeds only if the log only contains reading of grades, but regardless of the owner of the grades. This may be considered a kind of a *system* read access to grades.
- `readGrade s` is a TMI action that reads a grade of a student from the relevant `TMIVar` and returns it. The introspection log will contain an appropriate entry afterwards.
- `averageGrades` is a TMI action that reads grades of all students from the appropriate `TMIVars` and returns their average. The introspection log will contain an entry for every read grade.

Now it is possible to define the following TMI action that provides a student with her own grade as well as the average grades of all students.

```
gradeInformation :: Student -> TMI (Grade, Grade)
gradeInformation s =
  do own <- readGrade s
     avg <- liftSTM (authorized allGradesRead
                           averageGrades)
  return (own, avg)
```

This function may be called with the appropriate authorization function, namely one that only allows a student access to her own grades.

```
atomically (authorized (ownGradesRead s)
            (gradeInformation s))
```

By applying the operational semantics rules to this term, one can find that the innermost action `averageGrades` will be authorized by `allGradesRead` *before* turning it into an STM transition. This may, for example, happen through rule (*AURETI*). Note that such a rule does not keep the log entries of already authorized actions, i.e. the Σ is not affected in the \Rightarrow transition below the line. Thus the log of the nested action is not contained in the log authorized by the outer authorization function `ownGradesRead`.

This use of nesting constitutes a *privilege amplification* in a manner similar to stack inspection [Fournet and Gordon 2003].

5. Implementation

Our Haskell implementation is comprised of one module, `TMI`. The most important components 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 standard *writer monad* on top of the regular STM monad. The writer monad has the ability of collecting accumulating information in a sequence, which is exactly what we need to maintain the introspection log. In the `TMI` module, the regular STM module is imported under the name `T`.

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

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

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

For inserting entries in the log, we define the following shortcut, where `tell` is the standard function that the writer monad uses to collect information.

```
log :: (TMIAccess, d) -> TMI d ()
log entry = (TM . tell) [entry]
```

`log` returns an STM action which has the only effect of appending its argument to the introspection log. Note that this helper is not exported, so users of the `TMI` module cannot append to the log directly. As expected, a log entry of the form `(ReadVar, x)` just means that a variable with descriptor value `x` was read.

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 log (CreateVar, description)
     var <- liftSTM (T.newTVar val)
  return (TMIVar var description)
```

```
readTMIVar :: TMIVar d a -> TMI d a
readTMIVar tv =
  do log (ReadVar, getDescriptor tv)
     liftSTM (T.readTVar (getTVar tv))
```

```
writeTMIVar :: TMIVar d a -> a -> TMI d ()
writeTMIVar tv val =
  do log (WriteVar, getDescriptor tv)
     liftSTM (T.writeTVar (getTVar tv) val)
```

The combinators from the STM world are defined thus.

```
retryTMI = liftSTM T.retry

runTMI = runWriterT . unwrapTM  -- helper

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 `TMILog d -> Bool` and a TMI action; 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 straightforward.

```
authorized :: (TMILog d -> Bool) -> TMI d a
                                         -> T.STM a

authorized auth act = do
  (result, log) <- runTMI act
  if not (auth log)
    then throw (AssertionFailed "Access_denied")
    else return result
```

Note that a custom exception may be more appropriate but for sake of clarity we simply use the standard assertion failure to trigger a transaction abort.

Our Haskell TMI implementation can support variants of history-based policy enforcement [Abadi and Fournet 2003], in particular allowing *privilege amplification* with nested TMI actions as described in the previous section. In particular a call to `authorized` will return an STM action which *contains* a TMI operation and an associated authorization closure. When code inside a TMI action needs increased privileges it can nest a call to `authorized` with a different authorization manager and use `liftSTM` to lift the resulting STM operation back to the TMI level.

As in stack inspection [Fournet and Gordon 2003], privilege amplification provides TMI security managers with a useful escape hatch to perform operations as a more powerful “application principal”.

6. Discussion 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. The variants we experimented with in the design process did not always exhibit the behaviour that we expected or wanted. For this task, defining the formal semantics proved to be an essential tool to understand and evaluate different design decisions as well as spotting special cases that were not so obvious in the actual implementation. Indeed, constructing the semantics helped us discover bugs and unexpected behaviour in the code, even after weeks of

careful consideration. In addition the formal semantics gives a clear and unambiguous description of the TMI architecture.

The TMI architecture, as described in Section 3.1, can support the enforcement of stateful security policies that depend on the execution history over multiple transactions. In our paper [Birgisson et al. 2008], we have experimented with such policies in another implementation of the TMI architecture. However, we have not explored the addition of such facilities here, in order to simplify the exposition of our semantics and Haskell implementation.

The privilege amplification by nested TMI actions naturally relies on the programmer to ensure that the nested authorization manager does not violate the enclosing policy. The objective of the TMI architecture is to provide facilities for writing policy enforcement code, not to prevent injection of malicious code. In the case of library development where one deals with untrusted code, such nesting may not be desirable and can be disabled. We have experimented with other ways of implementing privilege amplification without relying on this nesting, with good results.

In our implementation we are maintaining the introspection 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 runtime library. Having the formal semantics as the definition of the desired behaviour should make such an implementation easier to construct and check.

Future work in this context also involves writing or porting complex software to the architecture to obtain realistic performance measurements. Also, our semantics may still be simplified, while allowing the same behavior; for example, the transaction log seems redundant in STM transitions, and may possibly be eliminated.

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.

Acknowledgments

We would like to thank Luca Aceto for his helpful comments on our use of Structural Operational Semantics. We also thank Wouter Swierstra and Josef Svenningsson for their detailed comments on our Haskell code. Finally we thank the anonymous reviewers, whose comments we have tried to address in this final version.

The first author was partly supported by the project “New Developments in Operational Semantics” (nr. 080039021) of The Icelandic Research Fund. Also, this research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme.

References

- M. Abadi and C. Fournet. Access control based on execution history. In *Networked and Distributed System Security Symposium*, 2003.
- Arnar Birgisson, Mohan Dhawan, Úlfar Erlingsson, Vinod Ganapathy, and Liviu Iftode. Security enforcement using software transactional memory. In *ACM Conference on Computer and Communications Security*, October 2008.
- Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/641909.641912>.
- Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.
- Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Conference on Principles and Practice of Parallel Programming*, 2005.
- Maurice Herlihy and J. Eliot B. Moss. Transactional support for lock free data structures. In *20th International Symposium on Computer Architecture*, June 1993.
- Maurice Herlihy, Victor Luchango, and Mark Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN OOPSLA*, Oct 2006.
- Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In M. Broy C. Hoare and R. Steinbrueggen, editors, *Engineering theories of software construction, Marktobendorf Summer School 2000*, pages 47–96. NATO ASI Series, IOS Press, 2001.
- Simon Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in haskell. May 2008.
- Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.
- Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, 1996.
- Gordon Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
- W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2), Mar 2008.