Chapter 9

Towards a Verified Implementation of Software Transactional Memory

Liyang HU[1], Graham Hutton[1]
Category: Research

Abstract: In recent years there has been much interest in the idea of concurrent programming using *transactional memory*, for example as provided in STM Haskell. While programmers are provided with a simple high-level model of transactions in terms of a stop-the-world semantics, the low-level implementation is rather more complex, using subtle optimisation techniques to execute multiple concurrent transactions efficiently, which is essential to the viability of the programming model.

In this article, we take the first steps towards a formally verified implementation of transactional memory. In particular, we present a stripped-down, idealised concurrent language inspired by STM Haskell, and show how a low-level implementation of this language can be justified with respect to a high-level semantics, by means of a compiler and its correctness theorem, mechanically tested using QuickCheck and the HPC (Haskell Program Coverage) toolkit. The use of these tools proved to be invaluable in the development of our formalisation.

9.1 INTRODUCTION

In recent years, traditional uniprocessors have reached a plateau in terms of raw operating speed, and we now have entered the era of multi-core processors [18]. However, traditional techniques for concurrent programming, in particular the use of explicit locks, are notoriously error-prone and hinder code reuse.

[1]School of CS, University of Nottingham, UK; {`lyh,gmh`}`@cs.nott.ac.uk`

One approach to addressing this problem is to adopt a lock-free model of concurrency and specify the desired behaviour of programs in a declarative manner, without requiring the programmer to be concerned about how this is achieved in practice. In the context of Haskell, this idea has been explored in the form of *software transactional memory* [7], in which sequences of read and write operations on memory can be specified to run atomically, in the sense that their intermediate states are not observable to other concurrent computations.

While atomicity provides programmers with a simple yet powerful mechanism to write concurrent programs, the actual implementation of this mechanism is rather more complex, using the notion of *transactions* [4] to better exploit the available multi-core hardware. In this article, we take the first steps towards a formally verified implementation of software transactional memory, inspired by STM Haskell. In particular, we make the following contributions:

- Identification of a simplified subset of STM Haskell and a semantics for this language, suitable for exploring design issues.
- A low-level virtual machine for this language in which transactions are made explicit, along with a semantics for this machine.
- A compiler from the language to the virtual machine, along with a correctness theorem, tested using QuickCheck [3] and HPC [5].

To the best of our knowledge, this is the first time that the correctness of a compiler for a language with transactions has been considered in a formal setting. This article is aimed at the functional programmers who are interested in the implementation and formalisation of software transactional memory. We require only a basic familiarity with Haskell, to the level of Bird's textbook [2]. An implementation of the model described in this paper may be found on the authors' websites.

9.2 STM IN HASKELL

Harris et al. [7] first introduced transactional memory to Haskell as an extension to the Glasgow Haskell Compiler. Notable is the fact that no modification to the Haskell language specification was necessary: its higher-order constructs are sufficient to implement the required control structures, whereas previous attempts [6] made changes to the language syntax.

The standard Haskell approach to sequencing read and write operations on memory is to use the IO monad to handle sequencing, and IORefs to represent mutable variables. STM Haskell provides analogous operations on TVars (transactional variables) within the STM monad:

$$
\begin{aligned}
(\gg\!=) &:: \mathsf{STM}\ \alpha \to (\alpha \to \mathsf{STM}\ \beta) \to \mathsf{STM}\ \beta \\
\mathit{return} &:: \alpha \to \mathsf{STM}\ \alpha \\
\mathit{newTVar} &:: \alpha \to \mathsf{STM}\ (\mathsf{TVar}\ \alpha) \\
\mathit{readTVar} &:: \mathsf{TVar}\ \alpha \to \mathsf{STM}\ \alpha \\
\mathit{writeTVar} &:: \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{STM}\ ()
\end{aligned}
$$

In this manner, the STM monad provides just the relevant operations within transactions (a term we use synonymously with 'STM action'), and precludes arbitrary and potentially irreversible side-effects. Executing transactions involves first converting them to IO actions using the *atomically* operation, and these actions can then be run concurrently using *forkIO* [16]:

```
atomically :: STM α → IO α
forkIO     :: IO α → IO ThreadId
```

Finally, STM Haskell supports a novel form of transaction composition, using a choice operator *orElse* that behaves as the first transaction if it succeeds and as the second transaction if the first fails, together with a *retry* primitive that forces failure:

```
orElse :: STM α → STM α → STM α
retry  :: STM α
```

Example

Suppose we represent the current balance of a bank account by a transactional variable *account* :: TVar Integer. Using the above operations, it is straightforward to implement a function that deposits a given amount:

```
deposit :: Integer → STM ()
deposit amount = do
  balance ← readTVar account
  writeTVar account (balance + amount)
```

Now if the following program is executed, the two deposits are guaranteed to take place, in either order. In particular, a deposit cannot be lost due to unexpected interleavings of *readTVar* and *writeTVar* in the definition of *deposit*.

```
do forkIO (atomically (deposit 10))
   forkIO (atomically (deposit 20))
```

9.3 A SIMPLE TRANSACTIONAL LANGUAGE

Our goal in this article is to formalise the low-level implementation details of software transactional memory. In order to focus on the essence of the problem, we abstract from the details of a real language such as STM Haskell, and consider a minimal language in which to explain and verify the basic implementation techniques. This section presents the syntax and semantics of our minimal language.

9.3.1 Language Syntax

The syntax of the language we consider can be defined by the following Haskell datatypes, where Tran represents transactions in the STM monad, Proc represents the desired aspects of concurrent processes in the IO monad, and Var represents a finite but unspecified collection of transactional variables:

$$\begin{array}{l} \textbf{data}\ \mathsf{Tran} = \mathsf{Val_T}\ \mathsf{Integer} \mid \mathsf{Tran} +_{\mathsf{T}} \mathsf{Tran} \mid \mathsf{Read\ Var} \quad\ \mid \mathsf{Write\ Var\ Tran} \\ \textbf{data}\ \mathsf{Proc} = \mathsf{Val_P}\ \mathsf{Integer} \mid \mathsf{Proc} +_{\mathsf{P}} \mathsf{Proc} \mid \mathsf{Atomic\ Tran} \mid \mathsf{Fork\ Proc} \end{array}$$

This language of expressions provides the essential computational features of STM Haskell, in a simplified form. On both levels, we replace sequencing ($\gg\!=$) and *return* with left-to-right addition ($+_.$) and integers. This has the advantage of avoiding the issues of name binding, yet still retains the fundamental monadic idea of sequencing computations and combining their results [11]. More formally, the use of integers and addition is justified by the fact that they form a monoid, a degenerate form of monads.

Read and Write are intended to mimic *readTVar* and *writeTVar*. We omit *newTVar* to once again avoid the issue of binding, and assume all variables are initialised to zero. Atomic runs a transaction to completion, delivering a value, while Fork spawns off its argument as a concurrent process, in the style of *forkIO*.

For simplicity, we do not consider *orElse* or *retry*, as they are not strictly necessary to illustrate the basic implementation of a log-based transactional memory system.

Example

Assuming a transactional variable *account* :: Var, our *deposit* function from the previous section can now be defined using our language as follows:

$$\begin{array}{l} \mathit{deposit} :: \mathsf{Integer} \to \mathsf{Tran} \\ \mathit{deposit}\ n = \mathsf{Write}\ \mathit{account}\ (\mathsf{Read}\ \mathit{account} +_{\mathsf{T}} \mathsf{Val_T}\ n) \end{array}$$

In turn, our example program of two concurrent deposits becomes:

$$\mathsf{Fork}\ (\mathsf{Atomic}\ (\mathit{deposit}\ 10)) +_{\mathsf{P}} \mathsf{Fork}\ (\mathsf{Atomic}\ (\mathit{deposit}\ 20))$$

9.3.2 Transactional Semantics

We specify the meaning of transactions in this language using a mostly small-step operational semantics, following the approach of [7, 15]. Formally, we give a reduction relation $\mapsto_{\mathsf{T}}$ on pairs $\langle h,\ e\rangle$ consisting of a heap h (a total map of type Var → Integer from variable names to their values) and a transaction expression e :: Tran. In this section we explain each of the inference rules defining $\mapsto_{\mathsf{T}}$.

First of all, reading a variable v looks up its value in the heap:

$$\langle h,\ \mathsf{Read}\ v\rangle \mapsto_{\mathsf{T}} \langle h,\ \mathsf{Val_T}\ h(v)\rangle \qquad \text{(READ)}$$

Writing to a variable is taken care of by two rules: (WRITE$\mathbb{Z}$) updates the heap with the new integer value for a variable in the same manner as the published semantics of STM Haskell [7], while (WRITET) allows its argument expression to be repeatedly reduced until it becomes a value:

$$\langle h,\ \mathsf{Write}\ v\ (\mathsf{Val_T}\ n)\rangle \mapsto_{\mathsf{T}} \langle h[v \mapsto n],\ \mathsf{Val_T}\ n\rangle \qquad (\text{WRITE}\mathbb{Z})$$

$$\frac{\langle h,\ e\rangle \mapsto_{\mathsf{T}} \langle h',\ e'\rangle}{\langle h,\ \mathsf{Write}\ v\ e\rangle \mapsto_{\mathsf{T}} \langle h',\ \mathsf{Write}\ v\ e'\rangle} \qquad (\text{WRITET})$$

Because we replace $\gg\!=$ with addition in our language, it is important to force a sequential evaluation order. The following three rules define reduction for $+_{\mathsf{T}}$, and ensure left-to-right evaluation:

$$\langle h,\ \mathsf{Val_T}\ m +_{\mathsf{T}} \mathsf{Val_T}\ n\rangle \mapsto_{\mathsf{T}} \langle h,\ \mathsf{Val_T}\ (m+n)\rangle \qquad (\text{ADD}\mathbb{Z}_{\mathsf{T}})$$

$$\frac{\langle h,\ b\rangle \mapsto_{\mathsf{T}} \langle h',\ b'\rangle}{\langle h,\ \mathsf{Val_T}\ m +_{\mathsf{T}} b\rangle \mapsto_{\mathsf{T}} \langle h',\ \mathsf{Val_T}\ m +_{\mathsf{T}} b'\rangle} \qquad (\text{ADDR}_{\mathsf{T}})$$

$$\frac{\langle h,\ a\rangle \mapsto_{\mathsf{T}} \langle h',\ a'\rangle}{\langle h,\ a +_{\mathsf{T}} b\rangle \mapsto_{\mathsf{T}} \langle h',\ a' +_{\mathsf{T}} b\rangle} \qquad (\text{ADDL}_{\mathsf{T}})$$

9.3.3 Process Semantics

The reduction relation $\mapsto_{\mathsf{P}}$ for processes acts on pairs $\langle h,\ s\rangle$ consisting of a heap h as before, and a 'soup' s of running processes [15]. The soup itself is a multi-set, which we represent as a list of type $[\mathsf{Proc}]$ for implementation reasons. The process rules are in general defined by matching on the first process in the soup. However, we begin by giving the (PREEMPT) rule, which allows the rest of the soup to make progress, giving rise to non-determinism in the language:

$$\frac{\langle h,\ s\rangle \mapsto_{\mathsf{P}} \langle h',\ s'\rangle}{\langle h,\ p : s\rangle \mapsto_{\mathsf{P}} \langle h',\ p : s'\rangle} \qquad (\text{PREEMPT})$$

Executing $\mathsf{Fork}\,p$ adds p to the process soup, and evaluates to $\mathsf{Val_P}\ 0$ (which corresponds to *return* () in Haskell) as the result of this action:

$$\langle h,\ \mathsf{Fork}\,p : s\rangle \mapsto_{\mathsf{P}} \langle h,\ \mathsf{Val_P}\ 0 : p : s\rangle \qquad (\text{FORK})$$

Next, the (ATOMIC) rule has a premise which evaluates the given expression until it reaches a value (where $\mapsto_{\mathsf{T}}^{\star}$ denotes the reflexive / transitive closure of $\mapsto_{\mathsf{T}}$), and a conclusion which encapsulates this as a single transition on the process level:

$$\frac{\langle h,\ e\rangle \mapsto_{\mathsf{T}}^{\star} \langle h',\ \mathsf{Val_T}\ n\rangle}{\langle h,\ \mathsf{Atomic}\,e : s\rangle \mapsto_{\mathsf{P}} \langle h',\ \mathsf{Val_P}\ n : s\rangle} \qquad (\text{ATOMIC})$$

In this manner we obtain a *stop-the-world* semantics for atomic transactions, preventing interference from other concurrently executing processes. Note that while the use of $\mapsto^{\star}_{\mathsf{T}}$ may seem odd in an otherwise small-step semantics, it expresses the intended semantics in a clear and concise manner [7].

Finally, it is straightforward to handle $+_{\mathsf{P}}$ on the process level using three rules, in an analogous manner to $+_{\mathsf{T}}$ on the transaction level:

$$\langle h,\ \mathsf{Val_P}\ m +_{\mathsf{P}} \mathsf{Val_P}\ n : s\rangle \mapsto_{\mathsf{P}} \langle h,\ \mathsf{Val_P}\ (m+n) : s\rangle \qquad (\text{ADD}\mathbb{Z}_{\mathsf{P}})$$

$$\frac{\langle h,\ b : s\rangle \mapsto_{\mathsf{P}} \left\langle h',\ b' : s'\right\rangle}{\langle h,\ \mathsf{Val_P}\ m +_{\mathsf{P}} b : s\rangle \mapsto_{\mathsf{P}} \left\langle h',\ \mathsf{Val_P}\ m +_{\mathsf{P}} b' : s'\right\rangle} \qquad (\text{ADDR}_{\mathsf{P}})$$

$$\frac{\langle h,\ a : s\rangle \mapsto_{\mathsf{P}} \left\langle h',\ a' : s'\right\rangle}{\langle h,\ a +_{\mathsf{P}} b : s\rangle \mapsto_{\mathsf{P}} \left\langle h',\ a' +_{\mathsf{P}} b : s'\right\rangle} \qquad (\text{ADDL}_{\mathsf{P}})$$

In summary, the above semantics for transactions and processes mirror those for STM Haskell, but for a simplified language. Moreover, while the original semantics uses evaluation contexts to identify the point at which transition rules such as (ADD$\mathbb{Z}_{\mathsf{P}}$) can be applied, our language is sufficiently simple to allow the use of explicit structural rules such as (ADDL$_{\mathsf{P}}$) and (ADDR$_{\mathsf{P}}$), which for our purposes have the advantage of being directly implementable.

9.4 A SIMPLE TRANSACTIONAL MACHINE

The (ATOMIC) rule of the previous section simply states that the evaluation sequence for a transaction may be seen as a single indivisible transition with respect to other concurrent processes. However, to better exploit the available multi-core hardware, an actual implementation of this rule would have to allow multiple transactions to run concurrently, while still maintaining the illusion of atomicity. In this section we consider how this notion of concurrent transactions can be implemented, and present a compiler and virtual machine for our language.

9.4.1 Instruction Set

Let us consider compiling expressions into code for execution on a stack machine, in which Code comprises a sequence of Instructions:

```
type Code        = [Instruction]
data Instruction = PUSH Integer | ADD | READ Var | WRITE Var
                 | BEGIN | COMMIT | FORK Code
```

The PUSH instruction leaves its argument on top of the stack, while ADD replaces the top two numbers with their sum. The behaviour of the remaining instructions is more complex in order to maintain atomicity, but conceptually, READ pushes the value of the named variable onto the stack, while WRITE updates the variable with the topmost value. In turn, BEGIN and COMMIT mark the start and finish of a transactions, and FORK executes the given code concurrently.

9.4.2 Compiler

We define the $\mathit{compile}_\mathsf{T}$ and $\mathit{compile}_\mathsf{P}$ functions to provide translations from Tran and Proc to Code, both functions taking an additional Code argument to be appended to the instructions produced by the compilation process; using such a *code continuation* both simplifies reasoning and results in more efficient compilers [10, §13.7]. In both cases, integers and addition are compiled into PUSH and ADD instructions, while the remaining language constructs map directly to their analogous machine instructions. The intention is that executing a compiled transaction or process always leaves a single result value on top of the stack.

```
compile_T :: Tran → Code → Code
compile_T e cc = case e of
  Val_T i    → PUSH i : cc
  x +_T y    → compile_T x (compile_T y (ADD : cc))
  Read v     → READ v : cc
  Write v e' → compile_T e' (WRITE v : cc)
compile_P :: Proc → Code → Code
compile_P e cc = case e of
  Val_P i    → PUSH i : cc
  x +_P y    → compile_P x (compile_P y (ADD : cc))
  Atomic e'  → BEGIN : compile_T e' (COMMIT : cc)
  Fork x     → FORK (compile_P x [ ]) : cc
```

For example, applying $\mathit{compile}_\mathsf{P}$ to our earlier program

```
compile_P (Fork (Atomic (deposit 10)) +_P Fork (Atomic (deposit 20))) [ ]
```

gives the following result:

```
[FORK [BEGIN, READ account, PUSH 10, ADD, WRITE account, COMMIT]
,FORK [BEGIN, READ account, PUSH 20, ADD, WRITE account, COMMIT]
,ADD]
```

9.4.3 Implementing Transactions

The simplest method of implementing transactions would be to suspend execution of all other concurrent processes on encountering a BEGIN, and carry on with the current process until we reach the following COMMIT. In essence, this is the approach used in the high-level semantics presented in the previous section. Unfortunately, this does not allow transactions to execute concurrently, one of the key aspects of transactional memory. This section introduces the log-based approach to implementing transactions, and discusses a number of design issues.

Transaction Logs

In order to allow transactions to execute concurrently, we utilise the notion of a *transaction log*. Informally such a log behaves as a cache for read and write operations on transactional variables. Only the first read from any given variable accesses the heap, and only the last value written can potentially modify the heap; all intermediate reads and writes operate solely on the log. Upon reaching the end of the transaction, and provided that that no other concurrent process has 'interfered' with the current transaction, the modified variables in the log can then be committed to the heap. Otherwise, the log is discarded and the transaction is restarted afresh.

Note that restarting a transaction relies on the fact that it executes in complete isolation, in the sense that all its side-effects are encapsulated within the log, and hence can be revoked by simply discarding the log. For example, it would not be appropriate to 'launch missiles' [7] during a transaction.

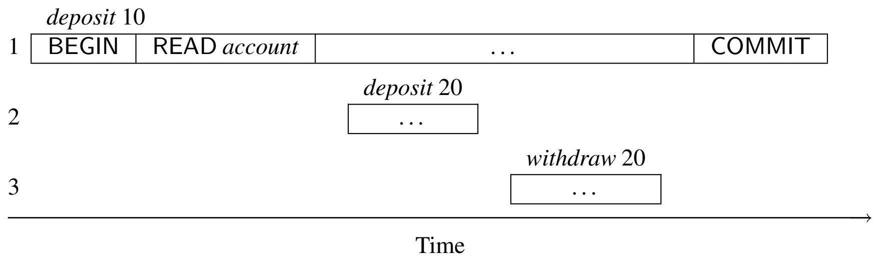
Interference

But what constitutes *interference*? When a transaction succeeds and commits its log to the heap, all of its side-effects are then made visible in a single atomic step, as if it had been executed in its entirety at that point with a stop-the-world semantics. Thus when a variable is read for the first time and its value logged, the transaction is essentially making the following bet: at the end of the transaction, the value of the variable in the heap will still be the same as that in the log.

In this manner, interference arises when any such bet fails, as the result of other concurrent processes changing the heap in a way that invalidates the assumptions about the values of variables made in the log. In this case, the transaction fails and is restarted. Conversely, the transaction succeeds if the logged values of all the variables read are 'equal' to their values in the heap at the end of the transaction.

Equality

But what constitutes *equality*? To see why this is an important question, and what the design choices are, let us return to our earlier example of a transaction that deposits a given amount into an account. Consider the following timeline:



Suppose that *account* starts with a balance of zero, which is read by the first transaction and logged. Prior to its final COMMIT, a second concurrent transaction successfully makes a deposit, which is subsequently withdrawn by a third transaction. When the first finally attempts to commit, the balance is back to zero as originally logged, even though it has changed in the interim. Is this acceptable?

i.e. Are the two zeros 'equal'? We can consider a hierarchy of notions of equality, in increasing order of permissiveness:

- The most conservative choice is to increment a global counter every time the heap is updated. Under this scheme, a transaction fails if the heap is modified at any point during its execution, reflected by a change in the counter, even if this does not actually interfere with the transaction itself.
- A more refined approach is provided by the notion of *version equality*, where a separate counter is associated with each variable, and is incremented each time the variable is updated. In this case, our example transaction would still fail to commit, since the two zeros would be have different version numbers, and hence considered different.
- For a pure language such as Haskell, in which values are represented as pointers to immutable structures, *pointer equality* can be used as an efficient but weaker form of version equality. In this case, whether the two zeros are considered equal or not depends on whether the implementation created a new instance of zero, or reused the old zero by sharing.
- We can also consider *value equality*, in which two values are considered the same if they have the same representation. In this case, the two zeros are equal and the transaction succeeds.
- The most permissive choice would be a *user-defined equality*, beyond that built-in to the programming language itself, in order to handle abstract data structures in which a single value may have several representations, e.g. sets encoded as lists. Haskell provides this capability via the Eq typeclass.

Which of the above is the appropriate notion of equality when committing transactions? Recall that under a stop-the-world semantics, a transaction can be considered to be executed in its entirely at the point when it successfully commits, and any prior reads are bets on the state of the heap at this point. Any intermediate writes that may have been committed by other transactions do not matter, as long as the final heap is consistent with the bets made in the log. Hence, there is no need at commit time to distinguish between the two zeroes in our example, as they are equal in the high-level expression language.

From a semantics point of view, therefore, value or user-defined equality are the best choice. Practical implementations may wish to adopt a more efficient notion of equality (e.g. STM Haskell utilises pointer equality), but for the purposes of this article, we will use value equality.

9.4.4 Virtual Machine

The state of the virtual machine is given by a pair $\langle h,\ s\rangle$, comprising a heap h mapping variables to integers, and a soup s of concurrent *threads*. Each Thread consists of a tuple of the form (c, σ, f, r, w), where c is the code to be executed, σ is the local stack, f gives the code to be rerun if a transaction fails to commit, and finally, r and w are two logs (partial maps from variables to integers) acting as read and write caches between a transaction and the heap.

$$\begin{aligned}
&\textbf{type}\ \mathsf{Thread} = (\mathsf{Code}, \mathsf{Stack}, \mathsf{Code}, \mathsf{Log}, \mathsf{Log})\\
&\textbf{type}\ \mathsf{Stack}\ \ = [\mathsf{Integer}]\\
&\textbf{type}\ \mathsf{Log}\quad\ = \mathsf{Var} \hookrightarrow \mathsf{Integer}
\end{aligned}$$

We specify the behaviour of the machine using a transition relation $\mapsto_{\mathsf{M}}$ between machine states, defined via a collection of transition rules that proceed by case analysis on the first thread in the soup. As with the previous semantics, we begin by defining a (PREEMPT) rule to allow the rest of the soup to make progress, giving rise to non-determinism in the machine:

$$\frac{\langle h,\ s\rangle \mapsto_{\mathsf{M}} \langle h',\ s'\rangle}{\langle h,\ t\!:\!s\rangle \mapsto_{\mathsf{M}} \langle h',\ t\!:\!s'\rangle} \qquad \text{(PREEMPT)}$$

This rule corresponds to an idealised scheduler that permits context switching at every instruction, as our focus is on the implementation of transactions rather than scheduling policies. We return to this issue when we consider the correctness of our compiler in §9.5.1.

Executing FORK adds a new thread t to the soup, comprising the given code c' with an initially empty stack, restart code and read and write logs:

$$\langle h,\ (\mathsf{FORK}\ c'\!:\!c,\ \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ 0\!:\!\sigma,\ f,\ r,\ w) : t : s\rangle \qquad \text{(FORK)}$$

where $t = (c',\ [\,],\ [\,],\ \emptyset,\ \emptyset)$

The PUSH instruction places the integer n on top of the stack, while ADD takes the top two integer from the stack and replaces them with their sum:

$$\begin{aligned}
&\langle h, (\mathsf{PUSH}\ n\!:\!c, \qquad \sigma,f,r,w)\!:\!s\rangle \mapsto_{\mathsf{M}} \langle h, (c, \qquad n\!:\!\sigma,f,r,w)\!:\!s\rangle \qquad \text{(PUSH)}\\
&\langle h, (\mathsf{ADD}\quad\!:\!c, n\!:\!m\!:\!\sigma,f,r,w)\!:\!s\rangle \mapsto_{\mathsf{M}} \langle h, (c, m+n\!:\!\sigma,f,r,w)\!:\!s\rangle \qquad \text{(ADD)}
\end{aligned}$$

Executing BEGIN starts a transaction, which involves clearing the read and write logs, while making a note of the code to be executed if the transaction fails:

$$\langle h,\ (\mathsf{BEGIN}\!:\!c,\ \sigma, f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ \sigma,\ \mathsf{BEGIN}\!:\!c,\ \emptyset,\ \emptyset) : s\rangle \qquad \text{(BEGIN)}$$

Next, READ places the appropriate value for the variable v on top of the stack. The instruction first consults the write log. If the variable has not been written to, the read log is then consulted. Otherwise, if the variable has also not been read, its value is looked up from the heap and the read log updated accordingly:

$$\langle h,\ (\mathsf{READ}\ v\!:\!c,\ \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ n : \sigma,\ f,\ r',\ w) : s\rangle \qquad \text{(READ)}$$

$$\text{where } \langle n,\ r'\rangle = \begin{cases} \langle w(v),\ r\rangle & \text{if } v \in \mathrm{dom}(w)\\ \langle r(v),\ r\rangle & \text{if } v \in \mathrm{dom}(r)\\ \langle h(v),\ r[v \mapsto h(v)]\rangle & \text{otherwise} \end{cases}$$

In turn, WRITE simply updates the write log for the variable v with the value on the top of the stack, without changing the heap or the stack:

$$\langle h,\ (\mathsf{WRITE}\ v\!:\!c,\ n\!:\!\sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h,\ (c,\ n\!:\!\sigma,\ f,\ r,\ w') : s\rangle \qquad \text{(WRITE)}$$

where $w' = w[v \mapsto n]$

Finally, COMMIT first checks the read log r for consistency with the current heap h, namely that the logged value for each variable read is equal to its value in the heap. Note that the write log may contain variables not in the read log, for which no check is necessary. Using our representation of logs and heaps, this condition can be concisely stated as $r \subseteq h$. If they are consistent, then the transaction has succeeded, so it may commit its write log to the heap. This update is expressed in terms of the overriding operator on maps as $h \oplus w$. Otherwise the transaction has failed, in which case the heap is not changed, the result on the top of the stack is discarded, and the transaction is restarted at f:

$$\langle h,\ (\mathsf{COMMIT} : c,\ n : \sigma,\ f,\ r,\ w) : s\rangle \mapsto_{\mathsf{M}} \langle h',\ (c',\ \sigma',\ f,\ r,\ w) : s\rangle$$
$$\text{where } \langle h',\ c',\ \sigma'\rangle = \begin{cases} \langle h \oplus w,\ c,\ n : \sigma\rangle & \text{if } r \subseteq h \\ \langle h,\ f,\ \sigma\rangle & \text{otherwise} \end{cases} \qquad \text{(COMMIT)}$$

There is no need to explicitly clear the logs in the above rule, since this is taken care of by the first instruction of f always being a BEGIN.

9.5 CORRECTNESS OF IMPLEMENTATION

As we have seen, the high-level semantics of atomicity is both clear and concise, comprising a single inference rule (ATOMIC) that wraps up a complete evaluation sequence as a single transition. On the other hand, the low-level implementation of atomicity using transactions is rather more complex and subtle, involving the management of read and write logs, and careful consideration of the conditions that are necessary in order for a transaction to commit. How can we be sure that these two different views of atomicity are consistent? Our approach to establishing the correctness of the low-level implementation is to formally relate it to the high-level semantics via a compiler correctness theorem.

9.5.1 Statement of Correctness

In order to formulate our correctness result, we utilise a number of auxiliary definitions. First of all, since our semantics is non-deterministic, we define a relation *eval* that encapsulates the idea of completely evaluating a process using our high-level semantics:

$$p\ \underline{eval}\ \langle h,\ s\rangle \quad \Leftrightarrow \quad \langle \emptyset,\ [p]\rangle \mapsto_{\mathsf{P}}^{\star} \langle h,\ s\rangle \not\mapsto_{\mathsf{P}}$$

That is, a process p :: Proc can evaluate to any heap h and process soup s that results from starting with the empty heap and completely reducing p using our high-level semantics, where $\not\mapsto$ expresses that no further transitions are possible. Similarly, we define a relation *exec* that encapsulates complete execution of a thread t :: Thread using our virtual machine, resulting in a heap h and a thread soup s:

$$t\ \underline{exec}\ \langle h,\ s\rangle \quad \Leftrightarrow \quad \langle \emptyset,\ [t]\rangle \mapsto_{\mathsf{M}}^{\star} \langle h,\ s\rangle \not\mapsto_{\mathsf{M}}$$

Next, we define a function $load :: \mathsf{Proc} \rightarrow \mathsf{Thread}$ that converts a process into a corresponding thread for execution, which comprises the compiled code for the process, together with an empty stack, restart code and read and write logs:

$$load\, p = (compile_{\mathsf{P}}\, p\, [\,], [\,], [\,], \emptyset, \emptyset)$$

Dually, we define a partial function $unload :: \mathsf{Thread} \hookrightarrow \mathsf{Proc}$ that extracts the resulting integer from a completely executed thread into our process language:

$$unload\ ([\,], [n], f, r, w) = \mathsf{Val}_{\mathsf{P}}\, n$$

Using these definitions, the correctness of our compiler can now be expressed by the following relational equation, in which ; denotes composition of relations, and the functions *load* and *unload* are viewed as relations by taking their graph:

Theorem 9.1 (Compiler Correctness).

$$eval \;=\; load \;;\; exec \;;\; (id \times map\ unload)$$

That is, evaluating a process using our high-level semantics is equivalent to compiling and loading the process, executing the resulting thread using the virtual machine, and unloading each of the final values.

The above theorem can also be split into two inclusions, where $\supseteq$ corresponds to soundness, and states that the compiled code will always produce a result that is permitted by the semantics. Dually, $\subseteq$ corresponds to completeness, and states that the compiled code can produce every result permitted by the semantics.

In practice, some language implementations are not complete with respect to the semantics for the language by design, because implementing every behaviour that is permitted by the semantics may not be practical. For example, a real implementation may utilise a scheduler that only permits a context switch between threads at particular intervals, rather than after every transition as in our semantics, because doing so would be prohibitively expensive.

9.5.2 Validation of Correctness

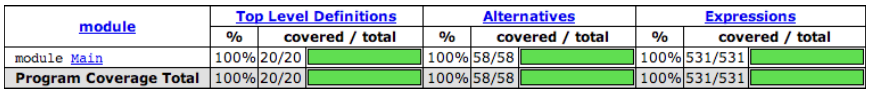
Proving the correctness of programs in the presence of concurrency is notoriously difficult. Ultimately we would like to have a formal proof of theorem 9.1, but to date we have adopted a mechanical approach to validating this result, using randomised testing.

QuickCheck [3] is a system for testing properties of Haskell programs. It is straightforward to implement our semantics, virtual machine and compiler in Haskell, and to define a property $prop\_Correctness :: \mathsf{Proc} \rightarrow \mathsf{Bool}$ that corresponds to theorem 9.1. Non-deterministic transitions in our system are implemented as set-valued functions, which are used to build up a tree that captures all possible evaluation sequences, thus ensuring all possible interleavings are accounted for. QuickCheck can then be used to generate a large number of random test processes, and check that the theorem holds in each one of these cases:

```
*Main> quickCheck prop_Correctness
OK, passed 100 tests.
```

Having performed many thousands of tests in this manner, we gain a high degree of confidence in the validity of our compiler correctness theorem. However, as with any testing process, it is important to ensure that all the relevant parts of the program have been exercised during testing.

The Haskell Program Coverage (HPC) toolkit [5] supports just this kind of analysis, enabling us to quickly visualise and identify unexecuted code. Using HPC confirms that testing our compiler correctness result using QuickCheck does indeed give 100% code coverage, in the sense that every part of our implementation is actually executed during the testing process:



module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module Main	100%	20/20	🟩	100%	58/58	🟩	100%	531/531	🟩
Program Coverage Total	100%	20/20	🟩	100%	58/58	🟩	100%	531/531	🟩

In combination, the use of QuickCheck for automated testing and HPC to confirm complete code coverage, as pioneered by the XMonad project [17], provides high-assurance of the correctness of our implementation of transactions.

9.6 CONCLUSION AND FURTHER WORK

In this article we have shown how to implement software transactional memory correctly, for a simplified language based on STM Haskell. Using QuickCheck and HPC, we tested a low-level, log-based implementation of transactions with respect to a high-level, stop-the-world semantics, by means of a compiler and its correctness theorem. This appears to be the first time that the correctness of a compiler for a language with transactions has been mechanically tested.

The lightweight approach provided by QuickCheck and HPC was indispensable in allowing us to experiment with the design of the language and its implementation, and to quickly check any changes. Our basic definitions were refined many times during the development of this work, both as a result of correcting errors, and streamlining the presentation. Ensuring that our changes were sound was simply a matter of re-running QuickCheck and HPC.

On the other hand, it is important to be aware of the limitations of this approach. First of all, randomised testing does not constitute a formal proof, and the reliability of QuickCheck depends heavily on the quality of the test-case generators. Secondly, achieving 100% code coverage with HPC does not guarantee that all possible interactions between parts of the program have been tested. Nonetheless, we have found the use of these tools to be invaluable in our work.

In terms of expanding on the work presented in this article, we have identified a number of possible directions for further work:

Proof. The most important step now is to consider how our correctness result can be formally proved. The standard approach [20] to compiler correctness for concurrent languages involves translating both the source and target languages into a common process language such as the π-calculus, where compiler correctness then amounts to establishing a bisimulation. We are in the process of developing a new, simpler approach that avoids the introduction of an intermediate language, by establishing a bisimulation directly between the source and target languages.

Generalisation. Our simplified language focuses on the essence of implementing transactions. However, it is important to take into account other features in the core language of STM Haskell, namely binding, input / output, exceptions and *retry* / *orElse*. Previous work by Huch and Kupke [9] describes a full implementation of the STM Haskell semantics given in [7], using existing Concurrent Haskell primitives, but they do not address the correctness of their implementation.

We could go further, and consider the implications of allowing limited effects within transactions, such as the creation of nested transactions or concurrent processes, with a view to investigate a more liberal variant of STM in Haskell.

Mechanisation. Just as QuickCheck and HPC were of great benefit for testing our compiler correctness theorem, we may similarly expect to benefit from the use of mechanical support when proving this result. Indeed, in the presence of concurrency it would not be surprising if the complexity of the resulting bisimulation proof necessitated some form of tool support. We are particularly interested in the use of automated proof-checkers such as Epigram [12] or Agda [14], in which the provision of dependent types allows proof to be conducted directly on the program terms, which helps to shift some of the work from the user to the type-checker [13]. Work on proving our correctness theorem in Agda is currently under way.

Other approaches. We have verified the basic log-based implementation of transactions, but it would also be interesting to consider more sophisticated techniques, such as suspending a transaction that has retried until a relevant part of the heap has changed. Finally, it is also important to explore the relationship to other semantic approaches to transactions, such as the use of functions [19] and processes [1], as well as relevant semantic properties, such as linearisability [8].

Acknowledgements

We would like to thank Nils Anders Danielsson, Peter Hancock, Tim Harris, James McKinna, Henrik Nilsson, Wouter Swierstra, the FP Lab at Nottingham, and the anonymous TFP referees for their useful comments and suggestions.

REFERENCES

[1] L. Acciai, M. Boreale, and S. D. Zilio. A Concurrent Calculus with Atomic Transactions. In *ETAPS Proceedings*. Springer-Verlag, April 2007.

[2] R. Bird. *Introduction to Functional Programming*. Prentice Hall, 2nd edition, 1998.

[3] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP Proceedings*, 2000.

[4] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition, 1995.

[5] A. Gill and C. Runciman. Haskell Program Coverage. In *Haskell Workshop Proceedings*, September 2007.

[6] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA Proceedings*, October 2003.

[7] T. Harris, S. Marlow, S. Peyton Jones, and M. P. Herlihy. Composable Memory Transactions. In *PPoPP Proceedings*, June 2005.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[9] F. Huch and F. Kupke. A High-Level Implementation of Composable Memory Transactions in Concurrent Haskell. In *Implementation and Application of Functional Languages, Lecture Notes in Computer Science*, volume 4015, pages 124–141, 2005.

[10] G. Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[11] G. Hutton and J. Wright. What is the Meaning of These Constant Interruptions? *Journal of Functional Programming*, 17(6):777–792, November 2007.

[12] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[13] J. McKinna and J. Wright. A Type-Correct, Stack-Safe, Provably Correct Expression Compiler in EPIGRAM. To appear in the Journal of Functional Programming, 2008.

[14] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, September 2007.

[15] S. Peyton Jones. Tackling the Awkward Squad. In *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.

[16] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL Proceedings*, pages 295–308, 1996.

[17] D. Stewart and S. Janssen. XMonad: A Tiling Window Manager. In *Haskell Workshop*, September 2007.

[18] H. Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr Dobb's Journal*, 30(3), March 2005.

[19] W. Swierstra. IOSpec: A Pure Specification of the IO Monad. Available from http://cs.nott.ac.uk/∼wss/repos/IOSpec/, 2008.

[20] M. Wand. Compiler Correctness for Parallel Languages. In *Functional Programming Languages and Computer Architecture*, pages 120–134, June 1995.