Transactional memory with data invariants

Tim Harris Simon Peyton Jones

Microsoft Research, Cambridge
{tharris,simonpj}@microsoft.com

Abstract

This paper introduces a mechanism for asserting invariants that are maintained by a program that uses atomic memory transactions. The idea is simple: a programmer writes `check E` where `E` is an expression that should be preserved by every atomic update for the remainder of the program's execution. We have extended STM Haskell to dynamically evaluate `check` statements atomically with the user's updates: the result is that we can identify precisely which update is the first one to break an invariant.

1. Introduction

Atomic blocks provide a promising simplification to the problem of writing concurrent programs [9]. A code block is marked `atomic` and the compiler and runtime system ensure that operations within the block, including function calls, appear atomic. The programmer no longer needs to worry about manual locking, low-level race conditions or deadlocks. Atomic blocks are typically built using *software transactional memory* (STM) which allows a series of memory accesses made via the STM library to be performed atomically.

This approach is sometimes described as being "like A and I" from ACID database transactions; that is, atomic blocks provide *atomicity* and *isolation*, but do not deal explicitly with *consistency* or *durability*. This paper attempts to include "C" as well, by showing how to define dynamically-checked data invariants that must hold when the system is in a consistent state. Specifically, we make the following contributions:

- We propose a simple but powerful new operation, `check E`, where `E` is an expression that must run without raising an exception after every transaction (Section 3). For example, given a predicate `isSorted` to test whether the data in a mutable list is sorted, an invariant `check (assert (isSorted l1))` would cause an error to be issued if any `atomic` block attempts to commit with the list `l1` unsorted. Furthermore, we can pinpoint exactly which `atomic` block attempted to violate the invariant.

  Using `atomic` blocks provides us with a key benefit over existing work on dynamically-checked invariants: the boundaries of `atomic` blocks indicate precisely where invariants must hold. They may, and often must, be broken *within* transactions, something that causes trouble in other systems (Section 7).

  Furthermore, the programmer has fine control over the granularity of invariant checking. She may specify coarse-grain invariants on large, global data structures, or fine-grain invariants on individual parts of those structures (e.g. Section 3.2).
- A distinctive feature of our work is that we give a complete, precise (but still compact) operational semantics of `check` in Section 4, by extending our earlier semantics for STM Haskell. This semantics gives a precise answer to questions such as: what happens if the invariant updates the heap, loops, or blocks?
- One might worry that, since invariants can be dynamically added but never deleted, the system will run slower and slower as more invariants are added. In Section 5 we show how to take advantage of the *existing* STM transaction logging mechanism to ensure that (*i*) invariants are only checked when a variable read by the invariant is written by a transaction, and (*ii*) invariants are garbage-collected entirely when the data structures they watch are dead. These properties are the key to scalability.
- In Section 6 we show how the operations supported by our invariants can be extended to express conditions relating pairs of program states ("`XYZ` is never decreased"), rather than just inspecting the current state ("`XYZ` is never zero").

The idea of combining data invariants with transactions is not new – indeed, the POSTQUEL query language from 1986 included a similar command that could be used to describe kinds of transaction that could not be committed against a database [24]. Section 7 discusses related work in that field, along with other work on incorporating invariants into programming languages.

We present our design in the context of STM Haskell [10] because this setting allows us to bring out the key issues in particularly crisp form. Everything we describe is fully implemented in the Glasgow Haskell Compiler, GHC, and will shortly be publicly available at the GHC home page. However, we believe that the ideas of the paper could readily be applied in other languages, as we discuss in Section 8.

2. Background: STM Haskell

Our prototype is based on STM Haskell [10], summarized in Figure 1. In this section we briefly review the language for the benefit of readers not already familiar with it.

STM Haskell is itself built on Concurrent Haskell [20] which extends Haskell 98, a pure, lazy, functional programming language. It provides explicitly-forked threads, and abstractions for communicating between them. These constructs naturally involve side effects which are accommodated in the otherwise-pure language a mechanism called *monads* [25]. The key idea is this: a value of type `IO a` is an "I/O action" that, when performed may do some input/output before yielding a value of type `a`. For example, the functions putChar and getChar have types:

```
putChar :: Char -> IO ()
getChar :: IO Char
```

That is, putChar takes a `Char` and delivers an I/O action that, when performed, prints the string on the standard output; while `getChar` is an action that, when performed, reads a character from the console and delivers it as the result of the action. A complete program must define an I/O action called `main`; executing the program means performing that action.

For example:

```
main :: IO ()
main = putChar 'x'
```

I/O actions can be glued together by a monadic bind combinator. This is normally used through some syntactic sugar, allowing a C-like syntax. Here, for example, is a complete program that reads a character and then prints it twice:

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

Threads in STM Haskell communicate by reading and writing transactional variables, or TVars. The operations on TVars are as follows:

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

All these operations make use of the STM monad, which supports a carefully-designed set of transactional operations, including allocating, reading and writing transactional variables. The readTVar and writeTVar operations both return STM actions, but Haskell allows us to use the same do {...} syntax to compose STM actions as we did for I/O actions. These STM actions remain tentative during their execution: in order to expose an STM action to the rest of the system, it can be passed to a function atomic, with type:

```
atomic :: STM a -> IO a
```

It takes a memory transaction, of type STM a, and delivers an I/O action that, when performed, runs the transaction atomically with respect to all other memory transactions. One might say:

```
main = do { ...; atomic (writeTVar r 3); ... }
```

Operationally, atomic takes the tentative updates and actually applies them to the TVars involved, thereby making these effects visible to other transactions. The atomic function and all of the STM-typed operations are built over the software transactional memory. This deals with maintaining a per-thread transaction log that records the tentative accesses made to TVars. When atomic is invoked the STM checks that the logged accesses are valid – i.e. no concurrent transaction has committed conflicting updates. If the log is valid then the STM commits it atomically to the heap. Otherwise the memory transaction is re-executed with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable guarantees: (*i*) only STM actions and pure computation can be performed inside a memory transaction; in particular I/O actions cannot; (*ii*) no STM actions can be performed outside a transaction, so the programmer cannot accidentally read or write a TVar without the protection of atomic. Of course, one can always write atomic (readTVar v) to read a TVar in a trivial transaction, but the call to atomic cannot be omitted.

As an example, this procedure atomically increments a TVar:

```
incT :: TVar Int -> IO ()
incT v = atomic (do x <- readTVar v
                    writeTVar v (x+1))
```

The implementation guarantees that the body of a call to atomic runs atomically with respect to every other thread; for example, there is no possibility that another thread can appear to read v between the readTVar and writeTVar of incT.

```
-- The STM monad itself
data STM a
instance Monad STM

-- Exceptions
throw :: Exception -> STM a
catch :: STM a -> (Exception->STM a) -> STM a

-- Running STM computations
atomic :: STM a -> IO a
retry  :: STM a
orElse :: STM a -> STM a -> STM a

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

Figure 1. The language level interface to transactional memory in STM Haskell

Although less relevant to our current paper, STM Haskell also provides facilities for *composable blocking*. The first construct is a retry operation:

```
retry :: STM a
```

The semantics of retry is to abort the current atomic transaction, and re-run it after one of the transactional variables it read from has been updated. For example, here is a procedure decT that decrements a TVar, but blocks if the variable is already zero:

```
dec :: TVar Int -> STM ()
dec v = do x <- readTVar v
           if x == 0
             then retry
             else writeTVar v (x-1)

decT :: TVar Int -> IO ()
decT v = atomic (dec v)
```

Finally, the infix orElse function allows two transactions to be tried in sequence: (s1 ‘orElse‘ s2) first attempts s1; if that calls retry, then s2 is tried instead; if that retries as well, then the entire call to orElse retries. For example, this procedure will decrement v1 unless v1 is already zero, in which case it will decrement v2 instead. If both are zero, the thread will block:

```
decPair :: TVar Int -> TVar Int -> IO ()
decPair v1 v2 = atomic (dec v1 ‘orElse‘ dec v2)
```

In addition, the STM code needs no modifications at all to be robust to exceptions. The semantics of atomic is that if the transaction fails with an exception, then no globally visible state change whatsoever is made.

Note that since our original paper on STM Haskell [10], we realized that the type ‘STM a’ might, more clearly, be called ‘Atomic a’ and that the function atomic could be renamed ‘perform’. The new names would make it clearer that operations such as readTVar and writeTVar are individual atomic actions that are combined monadically to form larger compound atomic actions, and also that perform is used only when actually making such a compound action visible to concurrent threads (rather than being necessary at every level when calling one transactional function from another). For consistency we are sticking with the published names, but mention the alternatives in case they help readers unfamiliar with the language.

3. The main idea

The main idea of the paper is to introduce a single new primitive

```
check :: STM a -> STM ()
```

Informally, check takes an STM computation that tests an invariant and, in addition, adds it to a global set of such invariants. At the end of *every* user transaction, *every* invariant in the global set must be satisfied if the user transaction is to be allowed to commit. If any invariant fails, indicated by throwing an exception, then the user transaction is rolled back and the exception propagates.

Since invariant checks are run repeatedly, and in an unspecified order, it is clearly desirable that they do not perform side effects or input/output. Our design partly offers this guarantee by construction: since the argument to check is an STM computation, the type system guarantees that it performs no input/output. Of course, as an STM computation, it can call writeTVar to attempt to update transactional memory – or, indeed, it can attempt any of the other actions in the STM monad. To avoid this kind of side-effect we use a fresh nested transaction to check each invariant *and then roll-back this transaction whether or not the invariant succeeds*. We give a fully-precise specification in Section 4, but first we discuss our design informally in the rest of this section.

In this section we introduce a number of examples showing how invariants can be defined. In many of our examples we use simple data structures built from TVars holding integer values. In Haskell, as in other languages, these examples could be written more generally to act across multiple types; we stick to integers for simplicity rather than due to limitations in the design or the implementation. For simplicity we also stick with straightforward imperative data structures.

3.1 Example 1: range-limited TVars

Consider the following example in which the type LimitedTVar holds a range-limited integer value. The function newLimitedTVar constructs a LimitedTVar with a specified limit. incLimitedTVar attempts to increment the value:

```
type LimitedTVar = TVar Int

newLimitedTVar :: Int -> STM LimitedTVar
newLimitedTVar lim =
  do { tv <- newTVar 0
     ; check (do { val <- readTVar tv
                 ; assert (val <= lim) })
     ; return tv }

incLimitedTVar :: Int -> LimitedTVar -> STM ()
incLimitedTVar delta tv
   = do { val <- readTVar tv
        ; writeTVar tv (val+delta) }
```

A key point is that the invariant is associated with the *creation* of the LimitedTVar, and not with its (perhaps diverse) *uses*. A programmer therefore can be confident that *every* LimitedTVar will *always* obey its invariant, rather than wondering whether perhaps one errant use has fallen though the net. The second key point is that the invariant is checked only at the *end* of (every) transaction; the invariant may temporarily be broken *during* a transaction. For example, a particular transaction may increase the variable beyond its limit provided that the same transaction decreases it again before the transaction ends. It is not useful, for example, to test the invariant every time the variable is written. Finally, it is worth noting that the invariant is a first-class closure; for instance it has a free variable lim that is not recorded in the LimitedTVar data structure at all.

An invariant may of course describe a *relationship* between mutable variables. For example, a limited TVar with a mutable limit might be described thus:

```
data LimitedTVarM
  = LTV { val :: TVar Int, limit :: TVar Int }
```

Now the invariant-check would read both the val and limit TVars, and compare them, failing if they do not stand in the desired relationship.

3.2 Example 2: a sorted list

Our second example illustrates the trade-offs involved in expressing the same invariant in different ways. Consider the following definition of a singly linked list of integers:

```
data ListNode
  = ListNode { val  :: TVar Int,
               next :: TVar (Maybe ListNode) }
```

Each ListNode holds a TVar Int which we will call the node's *value*, and a reference to a Maybe ListNode which we will call the *next node*. In Haskell, the type Maybe ListNode is essentially a nullable reference to a ListNode – its value is either Nothing (null), or Just l1 (a reference to l1). A Nothing *next node* indicates the end of the list.

If a list is to be held in sorted order then, informally, an invariant for all nodes could be “the next node is either null, or the next node's value is larger than this node's value”. This could be expressed as:

```
validNode :: ListNode -> STM ()
-- Throws exception for invalid node
validNode ListNode { val = v_val, next = v_next }
  = do { next_node <- readTVar v_next
       ; case next_node of                -- C1
           Nothing -> return ()
           ListNode { val = v_next_val } ->
             do { this_val <- readTVar v_val
                ; next_val <- readTVar v_next_val
                ; assert (this_val <= next_val) }
  }
```

Case statement C1 examines the contents of v_next: if it holds Nothing, then the invariant holds and we simply return; otherwise, the value fields of the two nodes are read and compared.

As with the first example, we could integrate this invariant with a function that constructs list nodes:

```
newListNode :: Int -> STM ListNode
newListNode val
 = do { v_val  <- newTVar val
      ; v_next <- newTVar Nothing
      ; let result = ListNode { val = v_val,
                                next = v_next }
      ; check (validNode result)
      ; return result }
```

This approach is effective if *all* ListNodes should occur in sorted lists. But perhaps some lists are sorted, and some are not – what then? In such cases the invariant could perhaps be expressed better as a property of a larger data structures:

```
validList :: ListNode -> STM ()
validList ln@(ListNode { next = v_next })
  = do { r <- validNode ln   -- Check first node
       ; next_val <- readTVar v_next
       ; case next_val of
           Nothing -> return ()
           Just ln' -> validList ln' }
```

The code instantiating the list can now assert that validList is always true, rather than expressing a per-node invariant.

The choice between these two approaches is largely a matter of taste and engineering. This example lets us raise two more issues beyond those already highlighted: (*i*) using per-node invariants enables more precise error reports ("node XYZ is out of order", versus "something in list ABC is out of order"), and (*ii*) in our implementation, per-node invariants may perform better: if the list is updated then only invariants in the vicinity of the update are re-checked, rather than the whole list being scanned.

3.3 Example 3: invariants over state pairs

Our third example illustrates a kind of invariants which *cannot* be expressed in STM Haskell. Suppose that we wish to create a *non decreasing* TVar, holding an integer value that is never allowed to be decreased by a transaction. We might attempt such a definition as follows:

```
newNonDecreasingTVar :: Int -> STM (TVar Int)
newNonDecreasingTVar val
 = do { r <- newTVar val
      ; p <- newTVar val
      ; check (do { c_val <- readTVar r
                  ; p_val <- readTVar p
                  ; assert (p_val <= c_val)
                  ; writeTVar p c_val  -- W1
                  })
      ; return r;
   }
```

The intention here is that r refers to the TVar holding the non-decreasing value, that p refers to r's previous value, and that the check ensures that the previous value is less than the current value. Unfortunately this will not work – the write at W1 that is responsible for recording the previous value will be rolled back each time the invariant is checked.

This example might make it appear tempting to allow some limited kind of updates to be made within invariant checks; there are many ways that the state modified by these updates could be kept distinct from the state visible to the application through its own TVars.

Leaving aside the question of exactly *how* updates are carried from one invariant check to another, retaining *any* kind of update is problematic semantically. This is because *running an invariant check is no longer an idempotent operation*. For instance, consider the following example in which the invariant check maintains a counter, failing when the counter reaches 10:

```
timebomb :: STM ()
timebomb
 = do { c <- newTVar 0
      ; check (do { c_val <- readTVar c
                  ; writeTVar c (c_val + 1)
                  ; assert (c_val < 10)
                  })
   }
```

What should this mean? Must the check be performed on every transaction (failing when exactly 10 have been committed)? May the invariant be checked multiple times on every transaction – after all, the invariant updates a TVar (c) that it itself depends on. Conversely, is it permitted to elide checking this invariant at all – after all, it is not associated with any data reachable by the application?

If such a definition is to be allowed then the only reasonable approach semantically would seem to be to execute it until it either fails or reaches a fixed point. This is not an attractive proposition in terms of performance and so *we do not provide any support for maintaining updates from one invariant check to another*.

Having said that, as we return to in Section 6, we can extend our system to support invariants such as newNonDecreasingTVar without allowing problems of the kind raised by timebomb.

3.4 Example 4: invariants as guards

Our final example illustrates a facet of our design on which we would particularly welcome feedback: what happens when an invariant blocks? Recall that in STM Haskell, blocking is expressed by a retry statement being executed inside an atomic block. Semantically, this aborts the block and re-executes it from the start, although the implementation delays this re-execution until one of the TVars read by the block has been updated (without such an update the block would simply retry again, spinning uselessly).

Suppose that we define a variant of the LimitedTVar type from Section 3.1 which blocks instead of failing (aside from naming, differences are highlighted in black):

```
newBlockingTVar :: Int -> STM LimitedTVar
newBlockingTVar lim =
  do { v_n <- newTVar 0
     ; always { val <- readTVar v_n
              ; if (val <= lim)
                   then return ()
                   else retry }
     ; return v_n }
```

The following atomic blocks create a TVar limited to 10, and then attempt to exceed that limit by incrementing it from 0 to 20:

```
xs <- atomic { newBlockingTVar 10 } -- A1
-- intervening code elided
atomic { incBlockingTVar 20 xs }  -- A2
```

What should this mean? One option is that it should simply be forbidden. An alternative option is that executing retry when checking an invariant is exactly the same as executing retry within the block being checked: A2 will block until the increment can succeed without breaching the limit (perhaps because of work done by a concurrent thread forked elsewhere).

Our current semantics and implementation follow the latter alternative. As we discuss in the next section it is debatable whether this is the best choice here; however, it is reminiscent of how the SCOOP concurrency extensions for Eiffel interpret method preconditions as blocking guard conditions [21].

3.5 Design choices

The preceding examples illustrated a number of decisions taken in the design of check. The first four of these are genuine design decisions on which we have selected one particular option based on the intuition gained from our examples:

[D1] The granularity at which invariants are checked coincides with transaction boundaries. This follows many designs for database invariants and, of course, it is necessary to allow such as "all entries in list L1 must also be in list L2" to be broken *inside* transactions that must update one list and then the other.

[D2] An invariant must succeed both when it is passed to check*, and also when the transaction proposing it is committed.* Our design follows that of many of the database systems in Section 7.2. Although the decision that invariants must succeed when passed to check is debatable, it is essential that any new invariants succeed at the end of the transaction proposing them. This allows future invariant failures to be correctly identify the offending transaction.

[D3] The check *function is an* STM *action, and so it can be composed with other* STM *actions in an atomic block.* An early design had check as an IO action, so that it could not be used within

atomic blocks. Our examples illustrate the benefit of having check be an STM action: it can be encapsulated in STM-typed constructor functions.

[D4] The closure passed to check *is itself an* STM *action: it proceeds by reading directly from the* TVars *the the invariant depends on.* This allows an invariant to re-use existing STM functions that may form part of the program logic.

However, beyond these basic decisions, there are a number of cases where clear guidance does not follow from simple examples. To a large extent these are cases that a 'well behaved' invariant should not exercise: what if it updates TVars rather than just reading them, what if it loops, or what if it calls retry, orElse, or even check?

We have explored two points in this design space. The first, in Section 3.6, is the one followed by our implementation and by Section 4's operational semantics. In this design we *do not* restrict the kinds of STM action that can be composed to form an invariant; instead we use nested transactions and roll-back to limit the kinds of side effect that can leak out from a badly behaved invariant. The second design, in Section 3.7 shows how we can use the Haskell type system to statically restrict invariants to only reading from TVars and performing pure computation.

3.6 Unrestricted invariants

Our first approach is to perform each invariant check in a *nested transaction*, and to roll back this nested transaction whether or not the invariant succeeds. This means that the invariant can use TVars internally without being able to affect the application's data structures.

This approach leads to the following behavior for 'badly behaved' invariants:

[D5] If an invariant does not terminate at the end of a transaction then the transaction does not terminate.

[D6] An invariant may update TVars *within its own execution.*

[D7] If an invariant evaluates to retry *then the user transaction is aborted and re-executed (potentially after blocking until it is worth re-executing it).*

[D8] If an invariant executes a check *statement, then the new invariant is checked at that point, but is not retained by the system.*

Some of these design choices are open for debate. Two particular examples are the use of retry within invariants and the use of ordinary (i.e. catchable) exceptions to indicate failures. Our example from Section 3.4 illustrates how an invariant incorporating retry can remove the need to repeat a guard condition across multiple atomic blocks.

We are somewhat uneasy with this kind of use. This is because it *requires* invariants to be checked at run-time: this is at odds with the intuition that testing could be disabled once a program appears to run without violations.

3.7 Restricted invariants

An alternative to the *unrestricted invariants* of Section 3.6 is to limit invariants to only reading from TVars. Doing so means that invariants cannot have side effects on TVars, or call retry, orElse, or check.

This kind of restriction can be elegantly integrated with the interface to transactional memory in STM Haskell. Figure 2 shows how. The STM type constructor gets an extra type argument, e, that characterises the effets in the computation. Specifically, a computation of type STM ReadOnly t performs only read effects, while one of type STM Full t has arbitrary STM effects. The types ReadOnly and Full are so-called *phantom types*; they have no data contructors and no values.

```
-- Phantom types for different kinds of STM action
data ReadOnly
data Full

-- The STM monad distinguishing between kinds
-- of STM action
data STM e a
instance Monad (STM e)

-- Exceptions
throw :: Exception -> STM e a
catch :: STM e a -> (Exception->STM e a) -> STM e a

-- Running STM computations
atomic :: STM Full a -> IO a
retry  :: STM Full a
orElse :: STM Full a -> STM Full a -> STM Full a

-- Transactional variables
data TVar a
newTVar   :: a -> STM Full (TVar a)
readTVar  :: TVar a -> STM e a
writeTVar :: TVar a -> a -> STM Full ()

-- Invariants
check :: STM ReadOnly a -> STM Full ()
```

Figure 2. The language level interface to transactional memory in STM Haskell, distinguishing between actions that can perform any STM action ("STM Full") and those that can only read from TVars ("STM ReadOnly").

The functions writeTVar, retry, and orElse in Figure 2 all return Full computations. In contrast, readTVar is polymorphic in e, and hence can be used in both ReadOnly and Full contexts. The operations return, (>>=), catch, and throw are all similarly polymorphic, and hence are usable in both contexts. The key funcion in Figure 2 is check: it takes a ReadOnly computation and returns a Full computation. So, for example, check (readTVar x) is well-typed, while check (retry) or check (writeTVar x v) is not.

This design has its attractions: read-only invariants may be more amenable to static verification, and the implementation does not need to track and roll-back their side effects. Conversely, restrictions limit the kinds of existing function that can be used in invariants – any algorithms that internally use TVars are prohibited, even if they do not clash with those used by the application. Furthermore, since executable invariants can still loop endlessly, it is not the case that check statements can be safely removed from an application once it runs without invariant failures.

4. Operational semantics

So far our discussion in Section 3 has been informal. It is hard to be sure that such descriptions cover all the combinations of these functions that might arise[1], so in this section we extend the formal, operational semantics of STM Haskell [10] to include the check primitive. We follow the design for *unrestricted invariants* from Section 3.6.

Figure 3 gives the syntax of a fragment of STM Haskell. Terms and values are entirely conventional, except that we treat the application of monadic combinators, such as return and catch, as

[1] As an example, even though we had completed a prototype implementation, the case of executing one invariant that proposes a second invariant is something we did not anticipate until writing these semantics.

$$
\begin{array}{rrcl}
 & x, y & \in & \mathit{Variable} \\
 & r, t & \in & \mathit{Name} \\
 & C & \in & \mathit{Constructor} \\
 & c & \in & \mathit{Char} \\
 & & & \\
\text{Value} & V & ::= & r \mid c \mid \backslash x \texttt{->} M \mid C\, M_1 \cdots M_n \\
 & & \mid & \texttt{return}\, M \mid M \texttt{>>=} N \\
 & & \mid & \texttt{putChar}\, c \mid \texttt{getChar} \\
 & & \mid & \texttt{throw}\, M \mid \texttt{catch}\, M\ N \\
 & & \mid & \texttt{retry} \mid M\ \texttt{`orElse`}\ N \\
 & & \mid & \texttt{forkIO}\, M \mid \texttt{check}\, M \\
 & & & \\
\text{Term} & M, N & ::= & x \mid V \mid M\ N \mid \cdots \\
 & & & \\
\text{Thread soup} & P, Q & ::= & M_t \mid (P \,|\, Q) \\
\text{Heap} & \Theta & ::= & r \hookrightarrow M \\
\text{Allocations} & \Delta & ::= & r \hookrightarrow M \\
\text{Invariants} & \Omega & ::= & \{M\} \\
 & & & \\
\text{Evaluation} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \texttt{>>=} M \\
\text{contexts} & & \mid & \texttt{catch}\, \mathbb{E}\, M \mid \mathbb{E}\ \texttt{orElse}\ M \\
 & \mathbb{P} & ::= & \mathbb{E}_t \mid (\mathbb{P} \,|\, P) \mid (P \,|\, \mathbb{P}) \\
\text{Action} & a & ::= & \texttt{!}\,c \mid \texttt{?}c \mid \epsilon
\end{array}
$$

Figure 3. The syntax of values and terms. Definitions in gray come directly from those used with STM Haskell. Definitions in black indicate modifications.

values. The `do`-notation we have been using so far is syntactic sugar for uses of `return` and `>>=`:

$$
\begin{array}{rcl}
\texttt{do}\,\{x\texttt{<-}e;\ Q\} & \equiv & e\ \texttt{>>=}\ \backslash x\texttt{->}\ \texttt{do}\,\{Q\} \\
\texttt{do}\,\{e;\ Q\} & \equiv & e\ \texttt{>>=}\ \backslash\_\texttt{->}\ \texttt{do}\,\{Q\} \\
\texttt{do}\,\{e\} & \equiv & e
\end{array}
$$

Figure 4 gives a small-step operational semantics for the language. Definitions typeset in gray are identical to the original definitions for STM Haskell. Definitions typeset in black show modifications or additions needed for `check`. We will first of all outline the structure of the definitions in this figure (Section 4.1) and then show how they are extended to support `check` (Section 4.2).

4.1 Original semantics

We begin by describing the operational semantics of STM Haskell without invariants. The material of this section is largely taken from [10], but it is essential to understanding the changes for invariants. The semantics is given in Figure 4, which groups the existing transitions into three sets:

The *IO transitions* are steps taken by threads. A transition $P; \Theta, \Omega \xrightarrow{a} Q; \Theta', \Omega'$ indicates a single step from a system with threads in state P transitions to one with threads in state Q. Theta (Θ) is the state of the heap before the transition; Θ' is the state of the heap after the transition. a is the IO action (if any) performed by the step. Omega (Ω) is the current set of invariants; we return to its role in section 4.2.

The first two rules deal with input and output. If the active term is a `putChar` or `getChar` the appropriate labelled transition takes place, and the operation is replaced by a `return` carrying the result. Rule FORK allows a new thread to be created, by adding a new term M to the thread soup, allocating a fresh name t as its `ThreadId`.

Rule ADMIN concerns *administrative transitions*, which are given in the second section of Figure 4. Rule EVAL allows a pure function M that is not a value to be evaluated by an auxiliary function, $\mathcal{V}[\![M]\!]$, which gives the value of M. This function is entirely standard, and we omit it here. Rule BIND implements sequential composition in the monad. The rules THROW, CATCH1 and CATCH2 implement exceptions in the standard way. All of these rules are, as we shall see, used both for IO transitions and STM transitions, which is why we keep them in a separate group.

Ignoring the additions for `check`, rules ARET and ATHROW define the semantics of atomic blocks that return a value ARET, or that throw an exception ATHROW. In each case the main idea is that the *only* way of performing "$\Rightarrow$" STM transitions is to package up the transitions for an entire atomic block and encapsulate them in a single "$\rightarrow$" IO transition; this is how atomicity is reflected in the rules.

An *STM transition* has the form $M; \Theta, \Delta, \Omega \;\Rightarrow\; N; \Theta', \Delta', \Omega'$. It defines a transition within a single thread from state M to N. Once again, Θ is the state of the heap and Ω holds the invariants that we return to in Section 4.2.

The role of delta (Δ) is more subtle: it records the *allocation effects* of the transition. For instance, rules READ, WRITE and NEW are concerned with primitive accesses to `TVars` and their main effect is to return a value from the heap ($\Theta(r)$ in READ), or to update the heap ($\Theta[r \mapsto M]$ in WRITE). However, notice that as well as adding a new mapping to Θ, NEW also adds it to Δ.

The reason for tracking allocation effects is the design choice that ATHROW *rolls back* the heap updates that a transaction makes when it terminates by an exception, but that it continues propagating the exception that caused the roll back. This exception may contain references to `TVars` that were allocated within the transaction and so we must retain these allocations if we are not to introduce dangling pointers. Δ collects up these allocation effects and the ATHROW rule constructs a new heap state by combining them with the previous heap state ($\Theta \cup \Delta'$).

The STM transition AADMIN incorporates pure computation, monadic bind and exception handling within transactions.

Finally, the three rules OR1, OR2 and OR3 define the `orElse` combinator. OR1 says that M_1 `'orElse'` M_2 behaves like M_1 if that returns a value. OR2 expresses says that if M_1 raises an exception then that forms the result of the `orElse` operation. OR3 says that if M_1 completes by calling `retry` then we try `M2` instead.

The alert reader may be wondering why there is no rule ARETRY to go along with ARET and ATHROW, to account for the fact that an STM computation may evaluate to `retry`. *There is no rule for this case.* What that means is that an atomic block in which all `orElse` choices end in `retry` cannot make a series of STM transitions that will allow the ARET or ATHROW rules to be applied. To make progress, another thread must be chosen.

4.2 Semantics of invariants

We are now ready to extend the semantics to incorporate `check`. There are three changes:

Firstly, the state associated with IO transitions and STM transitions now includes a set of invariants Ω. As Figure 4 shows, the majority of rules treat this set in the same way as the heap Θ.

Secondly, the STM transitions now include two rules for `check`. The first, CHECK1 is taken when the invariant holds at the point it is proposed. Above the line, the proposed invariant `M` evaluates to a `return` term in the current heap state. Below the line, the proposed invariant is added to Ω and the side effects of evaluating it are discarded. Note that the heap remains Θ and allocation effects Δ – even if M's execution allocates new `TVars` there is no way that they can leak out because the result N is discarded.

The second new STM transition, CHECK2, is taken when the invariant does not hold at the point it is proposed. Above the line, M evaluates to a `throw` term. Below the line, the exception is re-raised, rolling back any updates made by the failed `check` but

$$\boxed{\text{IO transitions} \qquad P;\Theta,\Omega \xrightarrow{a} Q;\Theta',\Omega'}$$

$$\begin{array}{rcll}
\mathbb{P}[\texttt{putChar}\ c];\ \Theta,\Omega & \xrightarrow{!c} & \mathbb{P}[\texttt{return}\ ()];\ \Theta,\Omega & & (PUTC) \\
\mathbb{P}[\texttt{getChar}];\ \Theta,\Omega & \xrightarrow{?c} & \mathbb{P}[\texttt{return}\ c];\ \Theta,\Omega & & (GETC) \\
\mathbb{P}[\texttt{forkIO}\ M];\ \Theta,\Omega & \rightarrow & (\mathbb{P}[\texttt{return}\ t] \mid M_t);\ \Theta,\Omega & t \notin \mathbb{P},\Theta,\Omega & (FORK)
\end{array}$$

$$\frac{M \ \rightarrow\ N}{\mathbb{P}[M];\ \Theta,\Omega \ \rightarrow\ \mathbb{P}[N];\ \Theta,\Omega}\ (ADMIN) \qquad \frac{M;\ \Theta,\{\},\Omega \overset{\star}{\Rightarrow} \texttt{throw}\ N;\ \Theta',\Delta',\Omega'}{\mathbb{P}[\texttt{atomic}\ M];\ \Theta,\Omega \rightarrow \mathbb{P}[\texttt{throw}\ N];\ (\Theta \cup \Delta'),\Omega}\ (ATHROW)$$

$$\frac{M;\ \Theta,\{\},\Omega \overset{\star}{\Rightarrow} \texttt{return}\ N;\ \Theta',\Delta',\Omega' \quad \forall M_i \in \Omega' : \big(M_i;\ \Theta',\{\},\{\} \overset{\star}{\Rightarrow} \texttt{return}\ N_i;\ \Theta'_i,\Delta'_i,\Omega'_i\big)}{\mathbb{P}[\texttt{atomic}\ M];\ \Theta,\Omega \rightarrow \mathbb{P}[\texttt{return}\ N];\ \Theta',\Omega'}\ (ARET\mathit{1})$$

$$\frac{M;\ \Theta,\{\},\Omega \overset{\star}{\Rightarrow} \texttt{return}\ N;\ \Theta',\Delta',\Omega' \quad \exists M_i \in \Omega' : \big(M_i;\ \Theta',\{\},\Omega' \overset{\star}{\Rightarrow} \texttt{throw}\ N_i;\ \Theta'_i,\Delta'_i,\Omega'_i\big)}{\mathbb{P}[\texttt{atomic}\ M];\ \Theta,\Omega \rightarrow \mathbb{P}[\texttt{throw}\ N_i];\ (\Theta \cup \Delta' \cup \Delta'_i),\Omega}\ (ARET\mathit{2})$$

$$\boxed{\text{Administrative transitions} \qquad M \ \rightarrow\ N}$$

$$M \quad \rightarrow \quad V \quad \text{if } \mathcal{V}[\![M]\!] = V \text{ and } M \not\equiv V \quad (EVAL)$$

$$\begin{array}{rcll}
\texttt{return}\ N \mathbin{\texttt{>>=}} M & \rightarrow & M\ N & (BIND) \\
\texttt{throw}\ N \mathbin{\texttt{>>=}} M & \rightarrow & \texttt{throw}\ N & (THROW) \\
\texttt{retry} \mathbin{\texttt{>>=}} M & \rightarrow & \texttt{retry} & (RETRY)
\end{array}
\qquad
\begin{array}{rcll}
\texttt{catch}\ (\texttt{return}\ M)\ N & \rightarrow & \texttt{return}\ M & (CATCH1) \\
\texttt{catch}\ (\texttt{throw}\ M)\ N & \rightarrow & N\ M & (CATCH2) \\
\texttt{catch}\ (\texttt{retry})\ N & \rightarrow & \texttt{retry} & (CATCH3)
\end{array}$$

$$\boxed{\text{STM transitions} \qquad M;\Theta,\Delta,\Omega \ \Rightarrow\ N;\Theta',\Delta',\Omega'}$$

$$\begin{array}{rcll}
\mathbb{E}[\texttt{readTVar}\ r];\ \Theta,\Delta,\Omega & \Rightarrow & \mathbb{E}[\texttt{return}\ \Theta(r)];\ \Theta,\Delta,\Omega & \text{if } r \in dom(\Theta) \quad (READ) \\
\mathbb{E}[\texttt{writeTVar}\ r\ M];\ \Theta,\Delta,\Omega & \Rightarrow & \mathbb{E}[\texttt{return}\ \ ()];\ \Theta[r \mapsto M],\Delta,\Omega & \text{if } r \in dom(\Theta) \quad (WRITE) \\
\mathbb{E}[\texttt{newTVar}\ M];\ \Theta,\Delta,\Omega & \Rightarrow & \mathbb{E}[\texttt{return}\ r];\ \Theta[r \mapsto M],\Delta[r \mapsto M],\Omega & \text{if } r \notin dom(\Theta) \quad (NEW)
\end{array}$$

$$\frac{M;\ \Theta,\{\},\Omega \overset{\star}{\Rightarrow} \texttt{return}\ N;\ \Theta',\Delta',\Omega'}{\mathbb{E}[\texttt{check}\ M];\ \Theta,\Delta,\Omega \Rightarrow \mathbb{E}[\texttt{return}\ \ ()];\ \Theta,\Delta,\big(\Omega \cup \{M\}\big)}\ (CHECK1)$$

$$\frac{M;\ \Theta,\{\},\Omega \overset{\star}{\Rightarrow} \texttt{throw}\ N;\ \Theta',\Delta',\Omega'}{\mathbb{E}[\texttt{check}\ M];\ \Theta,\Delta,\Omega \Rightarrow \mathbb{E}[\texttt{throw}\ N];\ \big(\Theta \cup \Delta'\big),\big(\Delta \cup \Delta'\big),\Omega}\ (CHECK2)$$

$$\frac{M \ \rightarrow\ N}{\mathbb{E}[M];\ \Theta,\Delta,\Omega \Rightarrow \mathbb{E}[N];\ \Theta,\Delta,\Omega}\ (AADMIN) \qquad \frac{M_1;\ \Theta,\Delta,\Omega \overset{\star}{\Rightarrow} \texttt{return}\ N;\ \Theta',\Delta',\Omega'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta,\Delta,\Omega \Rightarrow \mathbb{E}[\texttt{return}\ N];\ \Theta',\Delta',\Omega'}\ (OR1)$$

$$\frac{M_1;\ \Theta,\Delta,\Omega \overset{\star}{\Rightarrow} \texttt{throw}\ N;\ \Theta',\Delta',\Omega'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta,\Delta,\Omega \Rightarrow \mathbb{E}[\texttt{throw}\ N];\ \Theta',\Delta',\Omega'}\ (OR2) \qquad \frac{M_1;\ \Theta,\Delta,\Omega \overset{\star}{\Rightarrow} \texttt{retry};\ \Theta',\Delta',\Omega'}{\mathbb{E}[M_1\ \texttt{`orElse`}\ M_2];\ \Theta,\Delta,\Omega \Rightarrow \mathbb{E}[M_2];\ \Theta,\Delta,\Omega}\ (OR3)$$

Figure 4. Operational semantics of STM Haskell. Definitions in gray form the original semantics. Definitions in black show modifications.

keeping any allocation effects (Δ') that may be leaked by the exception.

Finally, in the IO transitions, there are substantial changes to ARET1 (for successful atomic blocks) and a new rule ARET2 (for atomic blocks that break an invariant). Aside from the updates to Ω, ARET1 adds an additional premise to the original rule: all of the invariants in place at the end of the atomic block must evaluate to `return` terms. Note that we consider all M_i in Ω' – this will pick up any new invariants added during the atomic block. Also, when evaluating each invariant, we discard the actual value returned and the updates that the invariant may make to the heap and to the set of invariants. This mirrors our informal notion that invariants are checked in nested transactions that are then rolled back.

The new rule ARET2 applies when any of the invariants evaluates to a `throw` term. As with ATHROW, the exception is propagated, retaining allocation effects but rolling back the remainder of the heap. Note that by using allocation effects Δ' and Δ'_i we retain any allocations in the original `atomic` block and any allocations made during the invariant's re-execution.

5. Implementation

We have implemented `check` as an extension to our existing prototype of STM Haskell [10, 11]. The main point of this section is to demonstrate that invariants can be implemented in a practical and scalable manner. At first sight one might have thought the opposite, because the specification requires that *every* invariant is checked after *every* `atomic` block, and that does not scale at all as the number of invariants grows. The main technical insight is that the *very same mechanism* that is already needed to support the STM (`atomic`, `retry`, `orElse` etc) can be re-used to trigger the checking of invariants: that is, an invariant INV is only run after a transaction T if a variable read by INV is written by T.

Is this technique actually consistent with the semantics of Figure 4? Note that rule ARET1 requires *all* invariants to complete successfully, whereas our implementation may skip the evaluation of an invariant that does not depend on a given `atomic` block. The worry is that the implementation may skip an invariant that does not terminate, allowing an `atomic` block to commit when rule ARET1 would not apply.

This is not a problem. In outline, suppose that an invariant `I1` would loop after an atomic block `A1`. If the set of `TVars` read by `I1` intersects the set updated by `A1` then our implementation will execute `I1` and the program will loop. Conversely, if the sets are disjoint then `I1`'s execution will not have affected by the atomic block and the looping would have occurred earlier (either after a block that did affect `I1`'s read set, or at the point `I1` was proposed).

In Section 5.1 we provide an overview of the original STM interface that we build on. We then discuss three steps in the implementation of `check`. The first step (Section 5.2) is how to identify the invariants that need to be checked at the end of an `atomic` block. The second (Section 5.3) is how to perform those checks. The third (Section 5.4) is how we extend `STMCommit` to ensure atomicity between the user's transaction and the checking of the invariants.

5.1 Original STM interface

The underlying STM is based on optimistic concurrency control: until it attempts to commit, a transaction builds up a private log recording the `TVars` that it has read from, the values that it has seen in the `TVars`, and the values that it proposes storing in them.

The commit operation itself is disjoint-access parallel [14] (meaning that transactions accessing non-overlapping sets of `TVars` can commit in parallel) and read-parallel [7] (meaning that a set of transactions that have read from, but not updated, a `TVar` can commit in parallel). The commit operation is built over per-`TVar` locks implemented as part of the Haskell runtime system. Locks are only held during commit operations. We considered using a non-blocking STM derived from Herlihy *et al.*'s design [12], Fraser's design [6] or Marathe *et al.*'s hybrid design [19]: the indirection provided by `TVars` provides a natural counterpart to the object handles that these STMs use. We chose the lock-based design for two reasons: (*i*) the implementation is simpler, and (*ii*) the Haskell runtime schedules Haskell threads between a pool of OS threads tuned to the number of available CPUs; this removes some of the importance of a non-blocking progress guarantee.

Within the multi-processor Haskell runtime system, the STM implementation provides an interface for managing transactions and performing reads and writes to `TVars`. The interface is shown in Figure 5. As usual, gray lines indicate existing parts of the interface and black lines indicate changes and additions[2].

```
// Basic transaction execution
TLog *STMStart()
TVar *STMNewTVar(void *v)
void *STMReadTVar(TLog *tlog, TVar *t)
void STMWriteTVar(TLog *tlog, TVar *t, void *v)

// Transaction commit operations
boolean STMIsValid(TLog *tlog)
boolean STMCommit(TLog *tlog)

// Nested-transaction operations
TLog *STMStartNested(TLog *outer)
void STMMergeNested(TLog *inner)


// Invariant management
List<Closure*> *STMGetInvariantsToCheck(TLog *tlog)
void STMDefineInvariant(TLog *tlog,
                        Closure *c, TLog *inner)
void STMRecordCheckedInvariant(TLog *outer,
                        Closure *c, TLog *inner)
```

Figure 5. The STM runtime interface

`STMStart` starts a new top-level transaction, returning a reference to its transaction log. `STMNewTVar`, `STMReadTVar` and `STMWriteTVar` provide the basic operations to create, read, and update transactional variables.

`STMIsValid` returns `True` if the specified transaction log is consistent with memory (transactions are periodically validated so that conflicts with concurrent transactions are guaranteed to be detected [10]). `STMCommit` attempts to commit the current transaction, return `True` if it succeeds and `False` otherwise.

`STMStartNested` creates a new transaction nested within the specified `outer` transaction. `STMMergeNested` attempts to commit a nested transaction by merging its transaction log into its parent's (the parent becomes invalid if the child was). Transaction logs are allocated in the garbage collected heap and remain private to a transaction until passed to `STMCommit`: a transaction is aborted by simply discarding all references to its log.

5.2 Identifying invariants to check

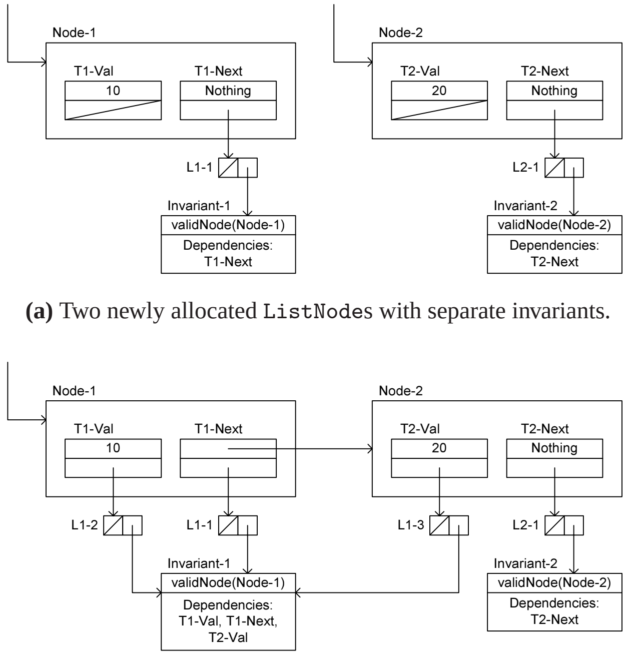
The key implementation idea is to dynamically track dependencies between invariants and `TVars`. We will illustrate this using the example in Figure 6(a). The figure shows two `ListNode` structures created by the `newListNode` function from Section 3.2. Each node comprises two `TVars`: one for its `val` field and one for its `next` field. The newly allocated nodes are not linked together, so the `next` fields both hold `Nothing`. Each `TVar` contains two fields: the first holds the `TVar`'s value and the second forms the head of a list of dynamic dependencies on the `TVar`. Link structures such as `L1-1` represent the dependencies between invariants and `TVars`[3]. For instance, `TVar T1-Val` has the value 10 and no dependents, whereas `T1-Next` has the value `Nothing` and is depended on by `Invariant-1`.

At runtime the invariants attached in `newListNode` are represented by structures holding the closure to be checked, and a list of the `TVars` that the invariant depended on when last evaluated. For instance, `Invariant-1` is evaluated by computing `validNode(Node-1)` whose result initially depends on `T1-Next` (because the current value of that `TVar` is `Nothing` and so the implementation of `validNode` does not examine the other `TVars`).

There are two sets of invariants to check at the end of an `atomic` block. Firstly, we must check any new invariants that

[2] For clarity we omit the further operations support blocking and unblocking Haskell threads that execute `retry` statements; these are unchanged and the details are orthogonal to this paper.

[3] As described in our earlier paper [10] the same list is used to represent dependencies between `TVars` and blocked transactions.



(a) Two newly allocated `ListNodes` with separate invariants.

(b) `Node-1` is updated to make `Node-2` its successor. This triggers re-evaluation of `Invariant-1` which checks that the two nodes are in order. `Node-1`'s invariant now depends on three `TVars`.

Figure 6. Runtime structures used to associate invariants with data that they depend on.

the block itself has proposed. Invariants are proposed by checking the invariant in a nested transaction, and if it succeeds, calling `STMDefineInvariant` which updates a new-invariant list attached to the current transaction log to include the supplied invariant and the dependencies established in its initial execution. Secondly, we must check any existing invariants that depend on `TVars` that the block intends to update. The function `STMGetInvariantsToCheck` in Figure 5 returns a single list containing both sources of invariants for the current transaction. Consider what happens when a transaction attempts to update `T1-Next` to link the two list nodes together – the update to `T1-Next` means that `STMGetInvariantsToCheck` just returns `Invariant-1`.

5.3 Checking invariants

Following the semantics of `check`, each invariant in the list returned by `STMGetInvariantsToCheck` must be confirmed to execute without raising an exception. This is done by iterating through the list and running each invariant in its own new transaction nested within the user's transaction.

If a check fails then the user's transaction is aborted and the exception indicating the failure is propagated[4]. If a check succeeds, then the invariant's closure and the nested transaction's log is passed to the STM through `STMRecordCheckedInvariant`. As described in the next section, the purpose of this call is to allow `STMCommit` to update the invariant's dependencies and to ensure that the whole set of invariant checks appear to take place atomically with the user's transaction.

```
10. Lock user-tlog tvars
   for each user-tlog log entry:
     if the entry is an update:
       try to lock the tvar
       if successful and current value matches entry:
         continue
       else:
         unlock tvars and abort
     if the entry is a read:
       record tvar's version number

15. Lock tvars related to invariants
   for each invariant touched
     for each tvar in current dependence set:  // I1
       try to lock the tvar
       if unsuccessful:
         unlock tvars and abort
     for each tvar in proposed dependence set: // I2
       try to lock the tvar
       if successful and current value matches that
                   read when checking the invariant:
         continue
       else:
         unlock tvars and abort

20. Check reads
   for each user-tlog entry:
     if the entry is a read then
       re-read the tvar's version number
       if this matches the one we recorded:
         continue
       else:
         unlock tvars and abort

25. Update invariant dependencies
   for each invariant touched                  // I3
     for each tvar in current dependence set:
       unlink tvar from invariant
     for each tvar in proposed dependence set:
       link tvar to invariant
     retain current dependence set as old set
     install proposed dependence set as current set

30. Make updates
   for each user-tlog entry:
     if the entry is an update:
       store new value to tvar, unlocking the tvar

35. Unlock tvars related to invariants
   for each invariant touched
     for each tvar in old dependence set:      // I4
       unlock the tvar if still locked
     discard old dependence set
     for each tvar in current dependence set:  // I5
       unlock the tvar if still locked
```

Figure 7. Committing a transaction with invariant checking.

5.4 Ensuring atomicity

We now consider the changes made to `STMCommit`. The underlying commit operation follows a pattern typical of many STM designs [7]: it acquires temporary ownership of the `TVars` that have been updated, it checks that `TVars` that have been read have not been modified by concurrent transactions, it applies the transaction's updates to the heap, and it finally releases ownership of the

[4] Unlike the operational semantics, our runtime system does not need to track the allocations that are made. This is because `STMNewTVar` places new `TVars` directly in the garbage collected heap.

TVars that it acquired. This is shown in the gray portions of Figure 7.

We extend this design with three additional steps shown in black in the figure. The inputs to these are the values passed to STMRecordCheckedInvariant, comprising the invariants' closures and the new dependence information from the transaction logs from the invariants' execution.

Step 15 ensures that STMCommit locks the TVars on which the invariant previously depended (loop I1), and the TVars it accessed when checked (loop I2). Note that some of these TVars may have already been locked in step 10, and that loop I2 must check the TVars' current values to ensure that the check is still up-to-date.

While holding these locks, step 25 updates the dependence information between the TVars and the invariants.

Finally, step 35 releases any locks that have not already been released in the existing step 30.

There are a number of design choices here. In particular, we chose to acquire *all* of the TVars in the dependence sets in loops I1 and I2. This serves two purposes: (*i*) the locks acquired in both loops protect the updates made in step 25, and (*ii*) the locks acquired in loop I1 also act as an implicit lock on the invariant. This is necessary to serialize concurrent user transactions attempting updates to distinct TVars on which the same invariant depends. An alternative design would explicitly lock invariants and use non-blocking lists to record the dependence between invariants and TVars. A non-blocking STMCommit algorithm could be developed by using helping in the usual way: all of the information needed by STMCommit is present at the start of the operation and can be made available through a descriptor in shared memory.

5.5 Garbage collection

The runtime structures in Figure 6 allow the memory occupied by invariants to be reclaimed automatically by the garbage collector: since there is no global list of invariants, each invariant becomes unreachable when all of the TVars it depends on become unreachable.

However, note that the links from invariants to TVars can extend the lifetimes of individual TVars that are not ordinarily reachable by the application. For instance, if T1-Val is reachable by the application then the dependency links through Invariant-1 will cause T1-Next and T2-Val (and everything reachable from them) to be retained even if the list nodes themselves are no longer reachable by the application.

6. Predicates over state pairs

Having seen this implementation, recall our problematic example from Section 3.3: what if we want to express a property over pairs of states ("XYZ never decreases") rather than a property of a single state ("XYZ is never zero")?

One could express such properties succinctly by allowing the invariant to read the "old" value of XYZ directly. Providing this ability is rather simple, because *the STM mechanism already retains* XYZ*'s old value in case the transaction is rolled back,* and so we can readily expose this value to the invariant check.

We can see two main approaches. The first is to provide a function to explicitly read the previous value from a TVar:

```
 readTVarOld :: TVar a -> STM a
```

However, while this is suitable for simple cases it requires separate functions to be used for access to the pre-transactional state. An alternative is to provide a mechanism for running an existing STM computation against the pre-transactional state:

```
 old :: STM a -> STM a
```

Using old we can express our example non-decreasing TVar as:

```
newNonDecreasingTVar :: Int -> STM TVar Int
newNonDecreasingTVar val
 = do { r <- newTVar val
      ; check (do { c_val <- readTVar r
                  ; p_val <- old (readTVar r)
                  ; assert (p_val <= c_val)
                  })
      ; return r;
   }
```

As with invariant checks in general, there are design choices to be made over what kinds of operations can be performed in an old computation. In fact, the same problems from Section 3.5 occur and, unsurprisingly, the two broad solutions from Section 3.6 and Section 3.7 are possible – that is, the old computation can either be run in its own transaction against the pre-transactional state, or the old computation can be statically restricted to just performing a series of readTVar operations. In the restricted setting we can give old the following type:

```
old :: STM ReadOnly a -> STM e a
```

As with check, this means that old can only be supplied with a ReadOnly STM action formed from readTVar operations and pure computation.

However, there are two additional problematic cases. Firstly, an old computation may try to read from a TVar that was allocated during the current transaction. This is straightforward to handle in our implementation because these *allocation effects* are kept distinct from the transaction's subsequent updates: the old computation will see the value with which the TVar was initialized.

The second problematic case is whether old should be usable *outside* an invariant check. Doing so could harm modularity because it allows an STM-typed function to depend on the starting state of the atomic block it occurs in, not just the state that it is called from. This is ultimately a matter of taste since there is no implementation reason to prevent such usage. However, if desired, we could restrct old to just being used in invariant checks by refining its type to:

```
old :: STM ReadOnly a -> STM ReadOnly a
```

The use of ReadOnly on the right hand side means that the action can *only* be performed in a context expecting a ReadOnly STM action – i.e. ultimately within an invariant check.

It is technically straightforward to add old to the semantics of Figure 4 but we omit the details because it is syntactically verbose: the state carried into and between STM transitions would have to include the pre-transactional state (Θ) captured in the ARET rules.

7. Related work

This paper builds on two main areas of existing work: (*i*) incorporating invariants in programming languages, and (*ii*) incorporating invariants in databases. We discuss these in Sections 7.1 and 7.2 respectively.

7.1 Invariants in programming languages

Many languages and tools have provided ways to express invariants over data. Gypsy and Alphard programs can include specifications for use by formal methods [8, 26]. CLU [18], ESC/Modula3 [4], ESC/Java [5] and JML [17] include specifications in stylized comments for processing by tools.

Euclid, Eiffel and Spec# are notable for embedding specifications in the same language that is used for programming. An important design decision in all of these languages is how to generalize invariants to be able to refer to multiple objects in the presence of aliasing. For instance, suppose that an invariant on a list states that it only contains positive-valued integers. It is insufficient to check this each time a node is added to the list because, in general, the contents of a node may subsequently be updated via another reference to it.

Euclid, Eiffel, Spec# and our own work all take different approaches to this problem. As we introduced in Section 1, a contribution of our approach is that we allow invariants to be *defined dynamically* (rather than, say, associated with class definitions), and that we allow them to depend on *arbitrary mutable state* (rather then, say, only on the fields of the current object).

Euclid includes explicit `assert` statements, pre- and post-conditions on routines, and invariants on modules[5] [16]. An invariant must remain true during the module's lifetime, except for when routines exported from the module are executing. Although these invariants could be written as boolean-typed Euclid expressions, they were generally expected to be checked by verification rather than checked at runtime [22] and so language mechanisms to control updates to data that an invariant depends on are not required.

The Eiffel language supports class-based invariants which must be satisfied by every instance of the class whenever the instance is externally accessible; that is, immediately after creation, and before and after any call to an exported routine of the class [13]. Invariants are boolean-typed Eiffel expressions. Note that invariants are explicitly checked *before* calls as well as after them: this will detect changes that may have been made to objects that the invariant depends on.

Spec# extends C# with several features to encourage robust programming [1]. These include class invariants that are required to hold on every instance of the class while it is not "exposed". A new construct `expose (o) { S }` allows the invariant of `o` to be temporarily broken within the statements `S`, but it must be restored by the end of those statements; objects can only be updated while exposed in this way. Furthermore, a hierarchical object-ownership discipline is used to ensure that the invariant of one object depends only on the state of that object and objects that it (transitively) owns. This means that an object's invariant cannot be broken by uncontrolled updates to objects that it depends on. In concurrent settings, the same hierarchy can be used to associate locks with aggregate objects [15].

7.2 Invariants in databases

Stonebraker introduced the idea of defining integrity constraints for a database independently from the basic requirements of its schema [23]. He described simple constraints on individual fields ("Employee salaries must be positive"), constraints on fields in the same row of a table ("Everyone in the toy department must make more than $8000"), and more complex constraints involving joins across tables ("Employees must earn less than two times the sales volume of their department if their department has a positive sales"). These constraints were expressed as a special form of query, and then enforced by combining them with database updates in such a way that an update cannot change data in a way that violates a constraint.

In the POSTQUEL query language, Stonebraker *et al.* introduced a more general system that supported integrity constraints and computation triggered by database updates [24]. Their system allowed existing commands to be tagged "always" or "refuse". An "always" command can be used to trigger updates when related data is modified, e.g. "Always replace Mike's salary with Bill's". Conceptually they run continuously: when first executed, the command runs until it ceases to have an effect, whereupon it is re-run whenever data that it has read or written to is updated. A "refuse" command can be used to enforce integrity constraints ("refuse to add an employee whose salary is more than $30k") or for security ("refuse to retrieve Mike's salary when logged in as Bill").

Cohen introduced "consistency rules" in the transactional lisp-derived query language AP5 [2]. This design is the closest to our own: all accepted transactions had to satisfy all of the constraints that were defined. Transactions were defined by series of queries grouped by an `atomic [ ... ]` construct; constraints could be violated within the atomic block, but had to be restored by the end of the block. Cohen's design allowed a user to specify whether or not a constraint had to be true at the point at which it was declared.

The SQL:92 query language supports various kinds of constraint definition [3]. In particular, *assertions* can be general constraints involving an arbitrary collection of columns from an arbitrary collection of tables. For instance, "no supplier with status less than 20 can supply any part in a quantity greater than 500":

```
CREATE ASSERTION supply CHECK
   ( NOT EXISTS ( SELECT * FROM S
                  WHERE S.STATUS < 20
                  AND EXISTS
                    ( SELECT * FROM SP
                      WHERE SP.SNO = S.SNO
                      AND SP.QTY > 500 ) ) )
```

Checking of constraints can be deferred within transactions and performed upon commit: if any constraint fails then the transaction fails and is rolled back.

8. Conclusion

The key ideas of this paper are to extend `atomic` blocks with a mechanism to dynamically define an invariant over arbitrary mutable state and to re-use the STM machinery to track the dependence between transactions and that state. The result is that the system provides the appearance that every committed `atomic` block preserves every invariant, while only re-evaluating invariants that a given block actually appears to have changed.

Some concluding observations:

Erasure. A frequent point of discussion about this work is whether invariants should be used to detect operations that are attempted when the system is 'not ready' for them – either indicating this explicitly by using `retry` within an invariant (as in Section 3.4), or by catching an exception raised by an invariant failure.

A possible benefit of this approach is code brevity: perhaps an application would include duplicate checks, one within the implementation of a transaction to check whether or not it is ready to run, and the second within an invariant attached to the data structures that are being modified.

Conversely, relying on invariants to control execution in this way makes it impossible to disable invariant-checking once a program has been debugged, and harms modularity because there is no external indication of whether or not a library operation requires invariant checking to be enabled.

This, we feel, provides a strong argument for keeping invariants for bug detection clearly distinct from similar operations that form part of the application's logic. An interesting approach (suggested by an anonymous reviewer) is to follow the database distinction between *assertions* and *triggers*: triggers are considered part of the application logic and may be used to maintain invariants between related data structures. In STM Haskell one could imag-

[5] In Euclid, `module` is a type constructor; many instances of a module can exist dynamically.

ine a trigger-like construct that could also use `retry` to defer the commit of a transaction when the system is not ready for it.

Expressiveness. We have shown how STM lets us extend invariant checks to include executable predicates over the *before* and *after* memory states of the transaction, rather than just the *after* state. This does raise the question of whether there are further kinds of invariant that would be useful to programmers but which cannot be expressed in our system. In principle there are some: nothing depending on three or more successive states can be expressed solely using invariant checks because any side effects incurred by checking invariants are rolled back.

We have considered one further possible design that increases the expressiveness of the properties that can be described solely by `check`s. The idea is to allow `check` statements to add new invariants to the system, even though we roll back ordinary updates that checks make to the heap. For instance, a 'non repeating `TVar`' that cannot take the same value more than once could be implemented by one invariant check that adds further checks each time a new value is seen. This is more expressive, but perhaps ultimately impracticable in many cases. There is one subtlety: any new invariants must themselves be checked against the post-transactional state as well as the state when `check` was called. This ensures that the complete set of invariants holds at the end of the transaction and that the set is closed under the re-execution of any invariant.

We have held back from actually implementing this more complicated design because, in practice, we think it is an open question as to whether there are *useful* properties that cannot be captured by our current design while still being suitable for expressing by executable specifications.

Application to other languages. It is easy to see how these ideas could be applied to a language other than STM Haskell. However, there are two issues that we would like to highlight. Firstly, our use of dynamically-defined invariants benefits from Haskell's support for closures: our examples in Section 3 showed how concise invariants depended on variables from enclosing scopes. Secondly, STM Haskell is notable in that the type system constrains where mutable state can be accessed: it is guaranteed that the *only* updates to transactional variables occur within `atomic` blocks. This lets us ensure that invariants are re-evaluated when necessary. In other languages it will be necessary to consider whether such a segregation is valuable.

Acknowledgments

The ideas in this paper have benefited greatly from discussion with the Spec# group and, in particular, we thank Daan Leijen, Mike Barnett and Ben Zorn for the the ideas of `readTVarOld`, `old`, and the use of phantom types.

References

[1] BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The Spec# programming system. In *Proceedings of CASSIS 2004* (2004).

[2] COHEN, D. Compiling complex database transition triggers. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1989), ACM Press, pp. 225–234.

[3] DATE, C. J., AND DARWEN, H. *A guide to the SQL standard*, 4th ed. Addison-Wesley, 2000.

[4] DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. Extended static checking. Tech. Rep. Research Report 159, Compaq SRC, Dec. 1998.

[5] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (New York, NY, USA, 2002), ACM Press, pp. 234–245.

[6] FRASER, K. *Practical lock freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2003.

[7] FRASER, K., AND HARRIS, T. Concurrent programming without locks. Under submission.

[8] GOOD, D. I., COHEN, R. M., AND HUNTER, L. W. A report on the development of Gypsy. In *ACM 78: Proceedings of the 1978 annual conference* (New York, NY, USA, 1978), ACM Press, pp. 116–122.

[9] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03)* (Oct. 2003), pp. 388–402.

[10] HARRIS, T., HERLIHY, M., MARLOW, S., AND PEYTON JONES, S. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, to appear* (June 2005).

[11] HARRIS, T., MARLOW, S., AND PEYTON JONES, S. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell* (Sept. 2005), pp. 49–61.

[12] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of distributed computing* (2003), ACM Press, pp. 92–101.

[13] INTERNATIONAL STANDARD ECMA-367, E. Eiffel analysis, design and programming language, June 2005.

[14] ISRAELI, A., AND RAPPOPORT, L. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1994), pp. 151–160.

[15] JACOBS, B., LEINO, R., AND SCHULTE, W. Safe concurrency for aggregate objects with invariants. In *Proceedings of SEFM 2005*.

[16] LAMPSON, B. W., HORNING, J. J., LONDON, R. L., MITCHELL, J. G., AND POPEK, G. J. Report on the programming language Euclid. *SIGPLAN Not. 12*, 2 (1977), 1–79.

[17] LEAVENS, G. T., RUBY, C., RUSTAN, K., LEINO, M., POLL, E., AND JACOBS, B. JML (poster session): notations and tools supporting detailed design in java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)* (New York, NY, USA, 2000), ACM Press, pp. 105–106.

[18] LISKOV, B. A history of CLU. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 1993), ACM Press, pp. 133–147.

[19] MARATHE, V. J., SCHERER III, W. N., AND SCOTT, M. L. Adaptive software transactional memory. Technical report TR-868, Department of Computer Science, University of Rochester, May 2005.

[20] MARLOW, S., PEYTON JONES, S., AND THALLER, W. Extending the Haskell Foreign Function Interface with concurrency. In *Proceedings of the ACM SIGPLAN workshop on Haskell* (Snowbird, Utah, USA, September 2004), pp. 57–68.

[21] MEYER, B. Systematic concurrent object-oriented programming. *Commun. ACM 36*, 9 (1993), 56–80.

[22] POPEK, G. J., HORNING, J. J., LAMPSON, B. W., MITCHELL, J. G., AND LONDON, R. L. Notes on the design of Euclid. In *Proceedings of an ACM conference on Language design for reliable software* (1977), pp. 11–18.

[23] STONEBRAKER, M. Implementation of integrity constraints and views by query modification. In *SIGMOD '75: Proceedings of the 1975 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1975), ACM Press, pp. 65–78.

[24] STONEBRAKER, M., AND ROWE, L. A. The POSTGRES papers. Tech. rep., Berkeley, CA, USA, 1987.

[25] WADLER, P. The essence of functional programming. In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992* (New York, NY, USA, 1992), ACM, Ed., ACM Press, pp. 1–14.

[26] WULF, W. A., LONDON, R. L., AND SHAW, M. An introduction to the construction and verification of Alphard programs. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering* (Los Alamitos, CA, USA, 1976), IEEE Computer Society Press, p. 390.