Master's Thesis

Michael Schröder

[draft 2015/03/30 15:08:00]

Contents

Chapter 1

Motivation

It is a widely held opinion that concurrent programming is difficult and error-prone. Low-level synchronization mechanisms, such as locks, are notoriously tricky to get right. Deadlocks, livelocks, heisenbugs and other issues encountered when writing complex concurrent systems are usually hard to track down and often confound even experienced programmers.

To simplify concurrent programming, higher-level abstractions are needed. One such abstraction is *Software Transactional Memory* (STM). Briefly, this technique allows the programmer to group multiple memory operations into a single atomic block, not unlike a database transaction. When implemented in a high-level language such as Haskell, with its emphasis on purity and its strong static type system, STM becomes especially powerful.

However, it is not always as powerful as we would like it to be. Disallowing arbitrary side effects, albeit for good reasons, limits STM's usefulness in certain situations. For example, while manipulating memory using STM is easy, persisting those manipulations in a transactionally safe way is frustratingly impossible. There is no way to achieve *durability*, an important component of database transactions, using pure STM. Another issue is *contention*. Some pure data types, such as maps, are highly inefficient when combined with STM, causing unreasonably high numbers of transactional conflicts. These could be avoided using local side effects.

In this thesis, I will explore the question: *How can we incorporate side-effecting actions into a transaction in a safe manner?* In particular, the contributions of my work are:

- A new STM primitive called atomicallyWithIO that allows the user to attach a *finalizer* to an STM transaction. Such finalizers are arbitrary I/O actions that can be used to serialize transactional data or incorporate external information into a transaction. I provide a detailed discussion of the semantics of this extension to STM, as well as my implementation of it in the Glasgow Haskell Compiler. (Chapter 2)

- A demonstration of the effectiveness of finalizers by constructing a thin

database abstraction on top of STM. I then use this to build a simple social networking site. (Chapter 3)

- A contention-free STM data structure, the *transactional trie*. It is based on the concurrent trie, but lifted into an STM context and combines transactions with carefully considered internal side effects. I present its design and implementation in Haskell, and evaluate it against other STM-specialized data structures. (Chapter 4)

In the remainder of this introductory chapter, I will give a brief overview of Haskell's STM interface.

1.1 STM in Haskell

Here are the main data types and operations of Haskell's STM interface:

```
data STM a
instance Monad STM
atomically :: STM a → IO a
data TVar a
newTVar :: a → STM (TVar a)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
retry :: STM a
orElse :: STM a → STM a → STM a
```

Atomic blocks in Haskell are represented by the STM monad. Inside this monad, we can freely operate on transactional variables, or TVars. We can read them, write them and create new ones. When we want to actually perform an STM computation and make its effects visible to the rest of the world, we apply atomically to the computation. This function turns an STM block into a transaction in the IO monad that, when executed, will take place atomically with respect to all other transactions.

For example, the following code snippet increments a transactional variable v:

```
atomically $ do x ← readTVar v
                writeTVar v (x + 1)
```

The use of atomically guarantees that no other thread can come in between the reading and writing of the variable. The sequence of operations happens indivisibly.

An important aspect of Haskell's STM implementation is that it is fully composable. Smaller transactions can be combined into larger transactions without having to know how these smaller transactions are implemented. An

important tool to make this possible is the composable blocking operator retry. Conceptually, retry abandons the current transaction and runs it again from the top. In the following example, the variable v is decremented, unless it is zero, in which case the transaction blocks until v is non-zero again.

```
atomically $ do x ← readTVar v
                if x ≡ 0
                  then retry
                  else writeTVar v (x − 1)
```

In addition to retry, there is the orElse combinator, which allows "trying out" transactions in sequence. m1`orElse`m2 first executes m1; if m1 returns, then orElse returns; but if m1 retries, its effects are discarded and m2 is executed instead.

STM is also robust against exceptions. The standard functions throw and catch act as expected: if an exception occurs inside an atomic block and is not caught, the transaction's effects are discarded and the exception is propagated.

Another interesting part of Haskell's STM are data invariants. Using the alwaysSucceeds function, one can introduce an invariant over transactional variables that is dynamically checked on every atomic update.

For more background on Haskell's STM, including its implementation, see the original STM papers (Harris, Marlow, et al. 2005; Harris and Peyton Jones 2006). For a more thorough exploration of not only STM but also other Haskell concurrency mechanisms, read Simon Marlow's excellent book on that topic (Marlow 2013a).

Chapter 2

Transactional Memory with Finalizers

2.1 STM and ACID

Atomicity, consistency, isolation and durability (ACID) are the cornerstones of any database system. Haskell's STM gives us three of these properties: transactions are always *atomic* and have an *isolated* view of memory; dynamically-checked invariants can be used to ensure that the state of the system is *consistent*. The one missing property is *durability*.

Making data durable means serializing it to storage. This is of course an I/O action, which is not allowed within the confines of STM. For good reason: STM transactions can retry at any time and I/O actions are in general neither idempotent nor interruptible. However, notice that for the purposes of durability it is not necessary to run arbitrary I/O at arbitrary moments during a transaction. We merely need to run a single I/O action to serialize our data at the end of the transaction.

The naive approach is to first atomically perform some STM computation and then independently serialize its result:

```
durably :: STM a → IO ()
durably m = do x ← atomically m
               serialize x
```

There are two issues with this. First, by virtue of the serialization happening independent of the atomic block, after we have performed a transaction another thread could perform a second transaction and serialize it before we have finished serializing the first one. Depending on our serialization method, we could end up with an inconsistent state between the memory of our program and what is stored on disk. At the very least, there is no guarantee that the ordering of events is preserved. Secondly, the function serialize might not terminate at all; it could throw an exception or its thread

could die. Again we would end up with an inconsistent state and possibly data loss.

So the serialization needs to be somehow coupled with the atomic transaction in such a way that the transaction only commits *after* the data has been serialized. We might try to use the ominously named unsafeIOToSTM function:

```
durably :: STM a → IO ()
durably m = atomically $ do x ← m
                            unsafeIOToSTM (serialize x)
```

The type of unsafeIOToSTM is IO $a \rightarrow$ STM a, effectively allowing us to circumvent the type system so we can unsafely perform I/O in the STM monad. But this too must fail: even if it is the last statement in the atomic block, serialize is performed before the transaction commits. The transaction might yet have to retry because of a conflict with another transaction. In doing so it also executes serialize again. Unless serialize is idempotent, this is certainly undesirable. Furthermore, if the thread receives an asynchronous exception, the transaction will abort in an orderly fashion, while serialize, with its irrevocable side effects, cannot be undone.

This leads us to the realization that *the transaction must only commit after the data has been serialized, and the data must only be serialized after the transaction has committed.* We can escape the paradox by clarifying the second restriction: the data can in fact already be serialized after the transaction is merely *guaranteed* to commit. Meaning: at the point of serialization, the transaction hasn't yet made its effects visible to other transactions, but there are no conflicts preventing it from doing so. And until serialization is finished there must be no way for any such conflicts to arise; for all intents and purposes, the transaction must count as committed, even though its effects are not visible yet and it could still roll back by its own volition (but not due to conflicts with other transactions).

What is needed, then, is a new STM primitive. Not a combination of existing functions, but a new fundamental operation that does exactly what we want. It would have to be implemented directly in the runtime system and we would have to make sure that it is sound and does not allow us to undermine the safety of STM in general.

I will now describe this primitive, give its detailed semantics and talk about my implementation of it in the Glasgow Haskell Compiler.

2.2 Finalizers

The main idea is to introduce a new function

```
atomicallyWithIO :: STM a → (a → IO b) → IO b
```

Like the existing atomically operation, atomicallyWithIO transforms a computation of type STM a into an I/O action. Additionally, it takes a *finalizer*, which is a function of type $a \to$ IO b, viz. an I/O action that can depend on the result of the STM computation, and combines it with the transaction in such a way that:

1. The finalizer is only performed if the STM transaction is guaranteed to commit.

2. The STM transaction only commits (i.e. makes its effects visible to other transactions) if the finalizer finishes without raising an exception.

A detailed specification of atomicallyWithIO will be given in Section 2.5. Everything described there has been fully implemented by me in GHC and I will discuss that implementation in Section 2.6. But first let us look at some motivating examples, which will demonstrate the usefulness of this new operation and highlight some of its finer issues.

2.2.1 Example 1: Printing tickets

Say we want to sell tickets for an event. We have a set number of tickets, stored in a transactional variable tickets of type TVar Int, and the following function to get the next available ticket:

```
nextTicket :: STM Int
nextTicket = do t ← readTVar tickets
                when (t ≡ 0) (error "sold out")
                writeTVar tickets (t − 1)
                return t
```

Our box offices can then use a statement like $t \leftarrow$ atomically nextTicket to safely acquire a ticket.

However, selling tickets involves actually printing the ticket number on a piece of paper. What happens if the printer malfunctions, or runs out of paper? If for some reason the ticket cannot be printed, we do not want that ticket number to be lost forever; it has not yet been sold after all. The solution, of course, is to use a finalizer:

```
printTicket :: Int → IO ()
printTicket t = ...
sellTicket :: IO ()
sellTicket = atomicallyWithIO nextTicket printTicket
```

If printTicket throws an exception, the transaction rolls back and the tickets counter was never decreased.

2.2.2 Example 2: User input

An I/O action can of course collect input as well. Consider the following example of an ATM. The function to withdraw money from an account is quite standard; as is, one would imagine, the function to dispense actual cash from the machine:

```
type Account = TVar Int
withdraw :: Account → Int → STM ()
withdraw acc amount = do bal ← readTVar acc
                         if bal < amount
                           then error "insufficient funds"
                           else writeTVar acc (bal − amount)
dispenseCash :: Int → IO ()
dispenseCash amount = ...
```

But instead of simply stringing these two functions together, we use a finalizer to ask the user for final confirmation before actually going through with the transaction:

```
getMoney :: Account → Int → IO ()
getMoney acc amount = atomicallyWithIO action finalizer
  where
    action = do withdraw acc amount
                bal ← readTVar acc
                return bal
    finalizer bal = do putStrLn ("New balance: " ++ show bal)
                       putStrLn "Confirm (yes/no)?"
                       answer ← getLine
                       case answer of
                         "yes" → dispenseCash amount
                         _     → error "cancelled"
```

What this example shows is how we can use atomicallyWithIO to play out the revocable effects of a transaction (e.g. withdrawing money from an account) and use the results in a side-effecting manner (e.g. showing the user the new balance of her account and asking for confirmation) before committing to those effects.

Note that the balance we show to the user will be the final balance. There is no way another transaction could change it before we finish the I/O action. Once the I/O phase begins, the transaction is locked. At this point, only the finalizer itself can cause a rollback. And only when the finalizer finishes by returning is the transaction made visible to the rest of the world.

There are obvious performance implications with this scheme: if the I/O action takes a long time, other transactions will have to wait before they

can make progress. This bottleneck is unavoidable, the transition from STM to I/O naturally creates a serialization point. It is up to the programmer to not waste undue time or otherwise ensure that the finalizer finishes in a timely manner, e.g. by using the standard timeout function.

2.2.3 Example 3: Nesting

Since the finalizer can be an arbitrary I/O action, the question arises: can we run atomicallyWithIO *inside* atomicallyWithIO?

Running atomically inside atomically is evidently not possible — the type system forbids it. Even something like

```
atomically (unsafeIOToSTM (atomically m))
```

won't work: GHC detects such nefariousness and throws a runtime exception. The runtime system does not support running a transaction inside another transaction, because there is no single intuitive way to resolve problems such as conflicting transactions or nested rollback. Luckily, it does not appear that this kind of nesting is necessary in practice or even useful.

But running atomically within a finalizer is a different story. First of all, this is eminently useful. Consider the following example, in which a counter is atomically increased only after confirmation has been received from two different servers. For efficiency, both servers are queried in parallel:

```
atomicallyWithIO (do n ← readTVar counter
                     writeTVar x (n + 1)
                     return n
                 )
                 (λn → do (r1, r2) ← concurrently (post n server1)
                                                  (post n server2)
                          if r1 ∧ r2
                            then return ()
                            else error "rejected"
                 )
```

The function concurrently :: IO a → IO b → IO (a, b) is exported by the async library (Marlow 2013b), which provides "a set of operations for running IO operations asynchronously and waiting for their results". The library is implemented using STM. When executing concurrently, there are actually multiple atomic transactions happening behind the scenes.

Many complex I/O actions might use STM internally.[1] There is no reason why these should not also be usable within the finalizer of atomicallyWithIO.

[1] Indeed, as of GHC 7.8, the base library itself makes use of STM in `GHC.Event.Unique`. This module is in turn used by I/O primitives like threadDelay. Thus STM usage is actually spread throughout large parts of the I/O subsystem.

The type system allows it, and the semantics are clear as well: finalizers are already regarded as irrevocable. A transaction run within the finalizer of another transaction is completely independent from that other transaction. There is only one restriction: *the inner transaction cannot modify any of the transactional variables used by the outer transaction*; this would inevitably lead to a deadlock, as the inner transaction would have to wait for the outer transaction to commit and release its locks on the shared variables, while the outer transaction can only commit once the inner transaction has committed.

But are these kinds of deadlock situations not exactly what STM is supposed to protect us from? True, but I feel that introducing the possibility of deadlocks — limited to this one specific scenario, in which the nested transactions operate on the same set of variables, which I believe is rather artificial — is justified by the great usefulness of allowing independent STM transactions to run during an STM finalizer. Also, the runtime system can always detect these kinds of deadlocks instead of simply looping forever, and it will throw an exception that gracefully aborts the transactions involved and that could be used to pinpoint the source of the bug.

In principle, circular dependencies between shared variables could even be detected statically, at compile time. Trying to enforce the non-circularity via the type system would probably involve advanced type system features implemented by GHC extensions, if not require dependent types outright. In any case, it would necessitate major backwards-incompatible changes to the existing STM API, which is something I wanted to avoid. Alternatively, one could imagine a separate pre-processing step in the compiler, but this seems rather inelegant and more trouble than it is worth.

We should note an important point here: a finalizer has the same global view of the world as any other I/O action running at the same time. In particular, the new value of a TVar updated during the STM part of the transaction will not be visible in the finalizer. The variable can be read safely – a deadlock only happens if an inner transaction tries to *modify* a shared variable – but the value returned will be its old value, which is the actual, globally known value of the TVar, at this moment in time.

2.3 Related Work

Extending Haskell's STM to allow safe combination of atomic blocks with I/O actions has been proposed from the very beginning. In a 2006 post on the `haskell-cafe` mailing list[2], Simon Peyton Jones suggested an operator

$$\mathsf{onCommit} :: \mathsf{IO}\ a \to \mathsf{STM}\ ()$$

[2] http://www.haskell.org/pipermail/haskell-cafe/2006-November/019771.html

that "would queue up an IO action to be performed when the transaction commits". The `stm-io-hooks` package (Robinson and Kuklewicz 2012) implements this operator in a custom STM monad that is meant as drop-in replacement for the system one. onCommit has the same semantics as atomicallyWithIO: "The commit IO action will be executed iff the transaction commits." The difference is that onCommit is truly composable. Any STM function can use onCommit to add an I/O action to the list of actions to be executed when the atomic block commits. The caller of atomically does not have to be aware of it.

While greater composability seems to be more in the spirit of Haskell's STM, I think that in this case it is actually dangerous. I/O is fundamentally not composable in the same way that STM is. Consider the following scenario:

```
foo :: STM ()
foo = do writeTVar a 1
         onCommit (serialize "a" 1)
bar :: STM ()
bar = onCommit (error "rollback")
baz :: IO ()
baz = atomically (foo ≫ bar)
```

Calling baz will lead to an inconsistent state: the onCommit action of bar will abort the whole transaction and so the STM effects of foo are rolled back, yet *the I/O effects are not*. What's worse, there is no way from looking only at baz to discern that any of this is going on. The composability of onCommit hides effects that in my opinion should be made explicit. This may not be an issue in all cases, but for serialization it clearly is. Using atomicallyWithIO is safer in this regard, at the cost of reduced composability.

2.4 Semantics of STM

I now formalize my design by giving an operational semantics. In this Section, I present an overview of the original semantics of STM Haskell as described by Harris, Marlow, et al. (2005), including the revised exception semantics from their post-publication appendix. Using this as a foundation, I will then add the changes necessary to implement finalizers in Section 2.5. There I will also include the invariant semantics described by Harris and Peyton Jones (2006).

Throughout my presentation of these semantics, I aimed to keep as close to the original papers as possible to ease cross-referencing. However, a few minor modifications have been made to better reflect the current state of GHC, such as changing the names of some of the primitives.

	x, y	$\in$	$Variable$
	r, t	$\in$	$Name$
	c	$\in$	$Char$
Value	V	$::=$	$r \mid c \mid$ `\` x `->` M
		$\mid$	`return` $M \mid M$ `>>=` N
		$\mid$	`putChar` $c \mid$ `getChar`
		$\mid$	`throw` $M \mid$ `catch` $M\ N$
		$\mid$	`retry` $\mid M$ `` `orElse` `` N
		$\mid$	`newTVar`
		$\mid$	`readTVar` $r \mid$ `writeTVar` $r\ M$
		$\mid$	`forkIO` M
		$\mid$	`atomically` M
Term	M, N	$::=$	$x \mid V \mid M\ N \mid \ldots$
Thread soup	P, Q	$::=$	$M_t \mid (P \mid Q)$
Heap	Θ	$::=$	$r \hookrightarrow M$
Allocations	Δ	$::=$	$r \hookrightarrow M$
Evaluation contexts	$\mathbb{S}$	$::=$	$[\cdot] \mid \mathbb{S}$ `>>=` M
	$\mathbb{P}$	$::=$	$\mathbb{S}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P})$
Action	a	$::=$	$!c \mid ?c \mid \epsilon$

Figure 2.1: The original syntax of STM Haskell (Harris, Marlow, et al. 2005)

Figures 2.1 to 2.4 give the complete syntax and semantics of the original version of STM Haskell. The key idea is to separate *I/O transitions* ("$\rightarrow$", Figure 2.3) from *STM transitions* ("$\Rightarrow$", Figure 2.4). Execution proceeds by non-deterministically choosing a thread and performing a single I/O transition. Thus, the execution of different threads may be interleaved and we have concurrency on the I/O level. However, STM transitions can not interleave. They are only ever performed as premises of the `atomically` operator and the resulting state change is thus regarded as a single atomic step on the I/O level.

Simply requiring that `atomically` must reduce (on the STM level) to either `return` or `throw`, and not `retry`, obviates the need for modeling transaction logs or rollback. These matters are left for the implementation.

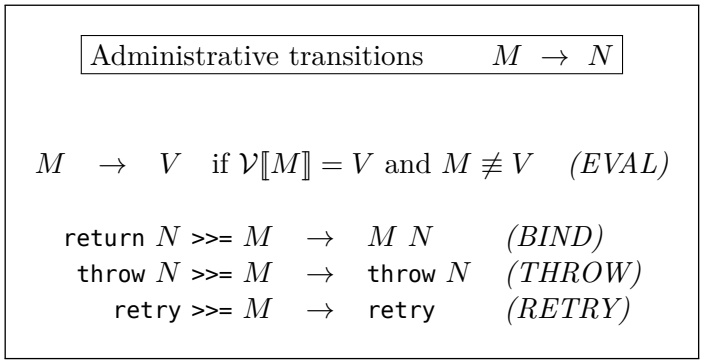

Figure 2.2: Administrative transitions of STM Haskell (Harris, Marlow, et al. 2005)

Syntax

The syntax of values and terms (Figure 2.1) is entirely conventional, except that some monadic combinators are treated as values. The `do`-notation used in the preceding sections is of course just the usual syntactic sugar for the monadic operators `>>=` and `return`:

```
do { x <- e; Q }  ≡  e >>= (\x -> do { Q })
     do { e; Q }  ≡  e >>= (\_ -> do { Q })
        do { e }  ≡  e
```

A *program state* $P;\Theta$ consists of a *thread soup* P and a *heap* Θ. The thread soup P is a multi-set of threads with each thread consisting of a single term M_t, where t is the thread ID. The heap Θ maps references to terms.

Additionally, we sometimes want to explicitly track a set of *allocation effects* Δ. This is a redundant subset of the heap. We mainly need it during exception handling, where we want to roll back heap effects but have to retain allocation effects.

When describing a transition between program states, we use an *evaluation context* to identify the active site of the transition. The program evaluation context $\mathbb{P}$ arbitrarily chooses a thread from the soup. This corresponds to the scheduler of a real implementation. STM terms are then evaluated using the context $\mathbb{S}$.

Administrative transitions

A few fundamental transitions are used on both the I/O and the STM level (Figure 2.2). The rule *(BIND)*, which implements sequential composition in the monad, as well as *(THROW)* and *(RETRY)* are quite ordinary. *(EVAL)*

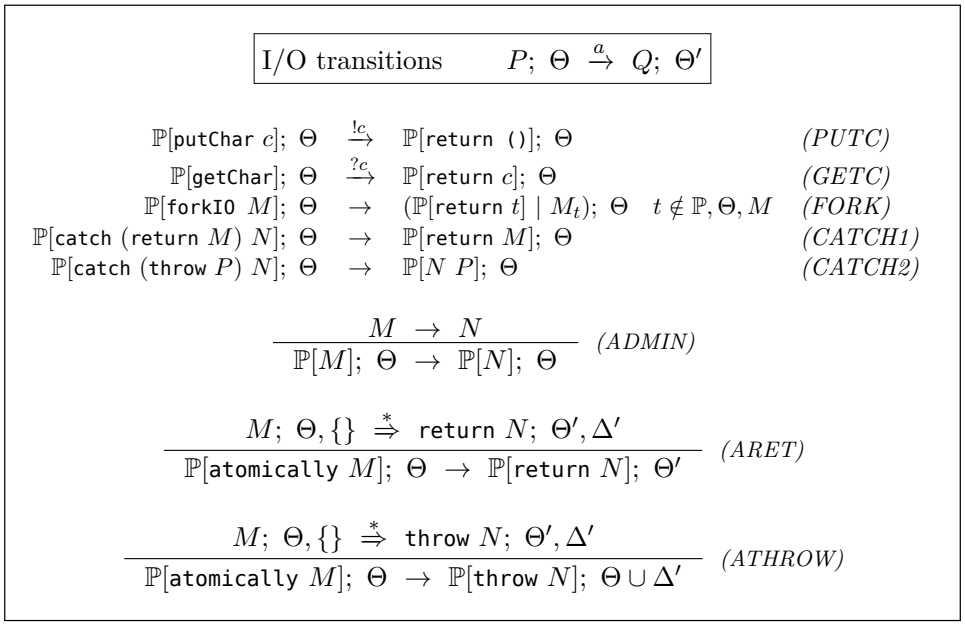

Figure 2.3: I/O-level transitions of STM Haskell (Harris, Marlow, et al. 2005)

allows evaluation of terms. It uses a function $\mathcal{V}$, which is entirely standard and whose definition is omitted.

The I/O level rule *(ADMIN)* lifts administrative rules into the I/O world. *(AADMIN)* does the same for STM.

I/O transitions

A top level I/O transition (Figure 2.3) is of the form

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

It takes a program state $P; \Theta$ to a new state $Q; \Theta'$ while performing the input/output action a. The actions $!c$ and $?c$ denote writing a character to the standard output and reading a character from the standard input, respectively. The silent action ϵ does nothing and is usually omitted. These are the only actions in our model, a real system would of course have many more.

The first two rules, *(PUTC)* and *(GETC)*, are applicable when the active term is `putChar` or `getChar`, respectively. The transition carries out the appropriate action and replaces the term with a `return` containing the result. The rule *(FORK)* creates a new thread, with a freshly chosen thread ID t, and adds it to the thread soup. *(CATCH1)* and *(CATCH2)* handle I/O level exceptions in the standard way.

More interesting are the rules *(ARET)* and *(ATHROW)*. They define the semantics of atomic blocks. We can see that the only way to perform zero or more STM transitions ("⇒") is by performing a single I/O transition ("→"), brought about by the `atomically` keyword. Interleaving of STM transitions is therefore not possible. Also, these are the only I/O level rules that affect the heap, which means that the heap can only be mutated inside an atomic block.

The rule *(ARET)* concerns STM terms that ultimately evaluate to `return` statements. This means the STM transaction was successfully completed and the updated heap Θ' becomes visible on the I/O level. If, on the other hand, the STM term evaluates to `throw` N, i.e. if the STM transaction throws an exception N, the exception is propagated to the I/O level and the new heap discarded. Only the allocation effects Δ' are kept and made globally visible. This is necessary because the exception value N may contain references to variables that were allocated inside the transaction and we do not want to leave dangling pointers around.

STM transitions

An *STM transition* (Figure 2.4) is of the form

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

Θ is again the heap, while Δ redundantly records the allocation effects. Note that none of these effects are visible globally until we make a transition on the I/O level that exposes them. All STM transitions stay completely within the STM layer, except for rule *(AADMIN)*, which just lifts pure administrative transitions.

Most of the STM rules are quite standard. *(READ)*, *(WRITE)* and *(NEW)* describe the reading, writing and allocating of mutable variables. Note how *(NEW)* makes allocation effects explicit by modifying Δ.

The rules *(OR1-3)* handle the kind of nested transactions made possible by the `orElse` combinator[3]. When encountering a term of the form M_1 `orElse` M_2 we always begin by inspecting the left branch M_1. If this evaluates to a `return` *(OR1)* or a `throw` *(OR2)*, the result is propagated and all memory effects are retained. Keep in mind that we are still on the STM level. If the whole STM transaction finally evaluates to `throw`, then its memory effects are of course globally discarded by the I/O level rule *(ATHROW)*. If we encounter a `retry` when evaluating M_1, rule *(OR3)* applies, discarding all memory effects and continuing with the evaluation of the right branch M_2.

Rules *(XSTM1-3)* describe exception handling within STM by use of the `catch` combinator. When evaluating the body M of a term `catch` M N,

[3] Not to be confused with the nesting of `atomically` calls described in Section 2.2.3

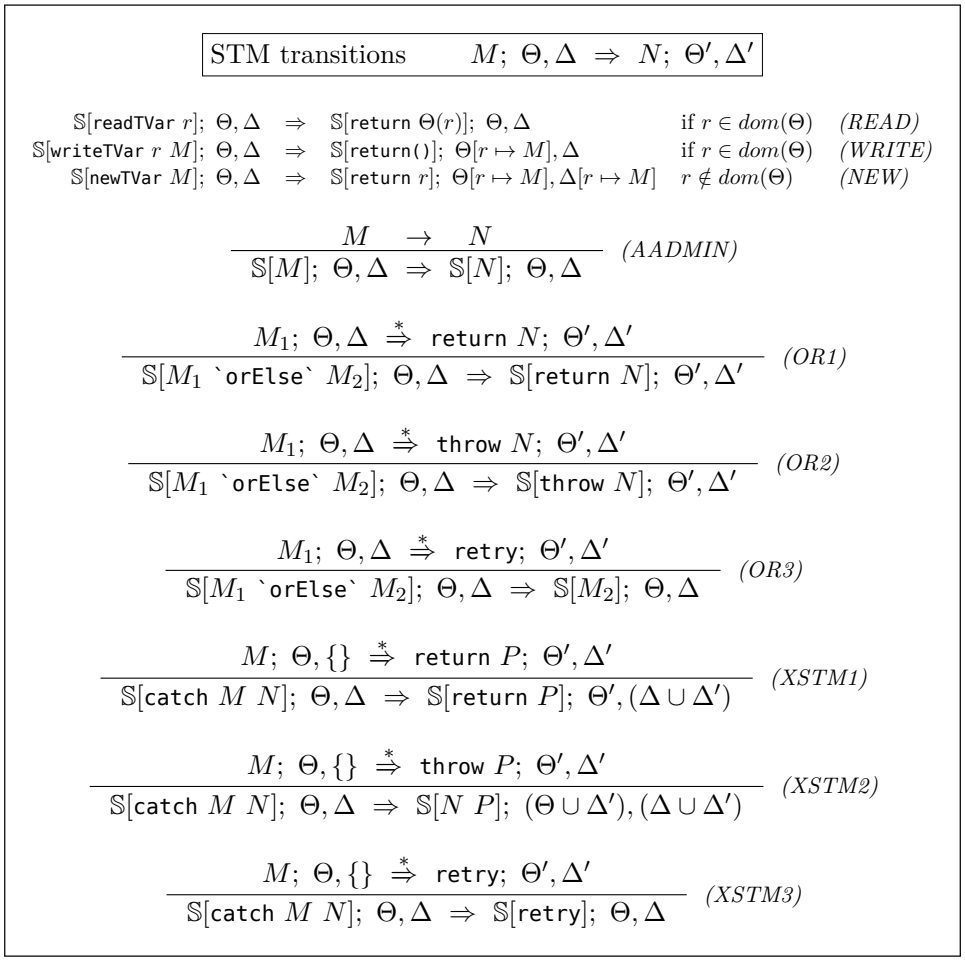

Figure 2.4: STM-level transitions of STM Haskell (Harris, Marlow, et al. 2005)

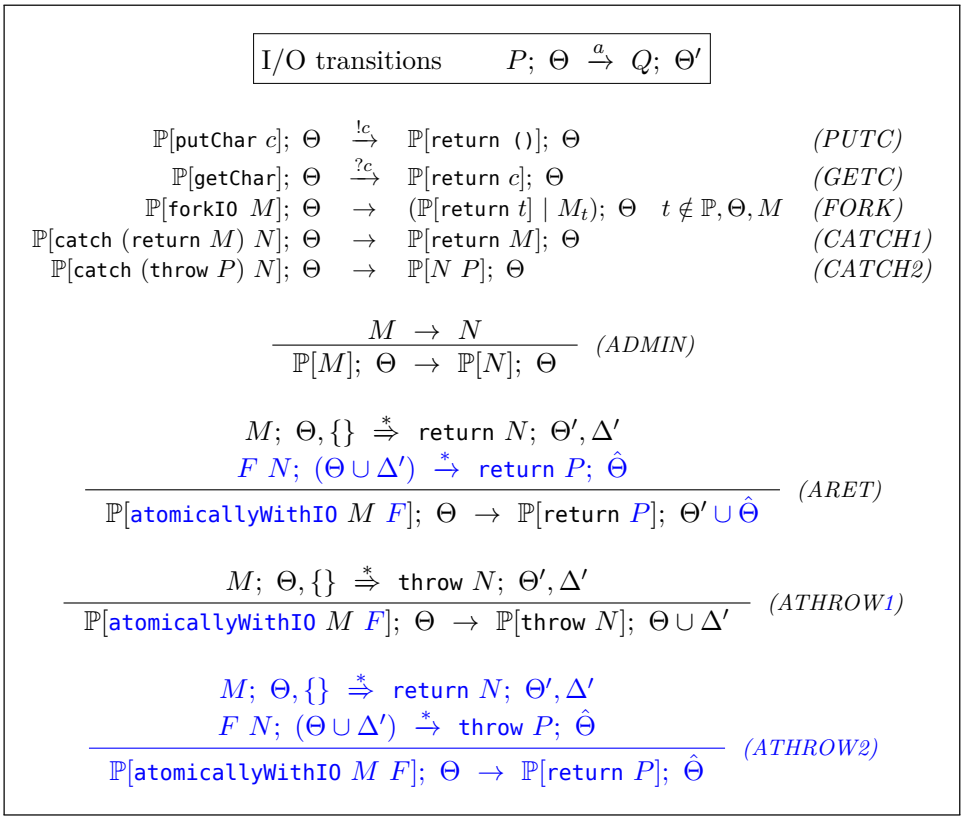

Figure 2.5: I/O-level transitions of STM Haskell with finalizers

we start with the current heap Θ and an empty set of allocation effects. On successful evaluation *(XSTM1)*, all effects are preserved and the catch handler N is ignored. If M throws an exception *(XSTM2)*, that exception is given to the handler N and only the allocation effects of M are preserved. If M evaluates to `retry` *(XSTM3)*, all effects are discarded.

This last set of rules was introduced in the post-publication appendix of Harris, Marlow, et al. (2005) to alleviate a subtle issue concerning leaking of effects. For a more elaborate discussion, of this issue and of the whole system, refer to that paper.

2.5 Semantics of finalizers

Now that we are familiar with the original STM semantics, we can extend them to support finalizers. First, let us realize that `atomically` is trivially expressed in terms of `atomicallyWithIO`:

```
atomically m  ≡  atomicallyWithIO m return
```

We can therefore use `atomicallyWithIO` to replace the old `atomically` completely. This keeps things simple and saves us from an annoying duplication of rules. Figure 2.5 gives the revised I/O transitions. There is no need to change anything on the STM level.

Rule *(ATHROW)* has been replaced by *(ATHROW1)*, which is identical except that it now uses `atomicallyWithIO`. Like before, if the STM term evaluates to an exception, the changed heap Θ' is discarded and only the allocation effects Δ' are kept. The finalizer F given to `atomicallyWithIO` is simply ignored in this case.

The heart of the change is to be found in rule *(ARET)*, which now has an additional premise: After the atomic block returns, the finalizer given to `atomicallyWithIO` must also evaluate to `return`. It does so by making zero or more *I/O-level* transitions. Note that these transitions begin with the original unchanged heap Θ. The finalizer executes in a global context, where the results of the STM transaction are not yet visible. Thus the finalizer cannot know about the modified heap Θ'. (It must know about the allocation effects Δ', however: N could contain a reference to a newly allocated variable and we don't want any dangling pointers.) The final heap resulting from the *(ARET)* transition is the union of the STM result heap Θ' with the result heap of the I/O transition, $\hat{\Theta}$.

The new rule *(ATHROW2)* handles exceptions that occur during the coupled I/O transition. Like with *(ATHROW1)*, the STM result heap Θ' is discarded. However, the I/O result heap $\hat{\Theta}$ *must be kept*, because it could have already been seen by other I/O level transitions. This is what we mean when we speak of irrevocable I/O.

2.5.1 Nesting

The attentive reader might have noticed that in the extended semantics given above the following term is perfectly valid:

$$\texttt{atomicallyWithIO}\ M_1\ (\texttt{atomicallyWithIO}\ M_2\ F)$$

As I have discussed in Section 2.2, this kind of nesting is actually pretty useful in practice. However, it does open up the possibility of deadlocks if the term M_2 writes to any of the memory locations referenced by M_1. Such behavior should be explicitly prohibited in the semantics, which need to be extended a bit further to do so (Figures 2.6 and 2.7).

Firstly, both I/O and STM transitions will carry around a new piece of state Σ, which is a set of labeled memory references:

$$\Sigma ::= r \times (\textsc{r} \mid \textsc{w})$$

It is a record of all memory operations done by a transaction. The labels R and W denote whether a particular memory location was read from or written

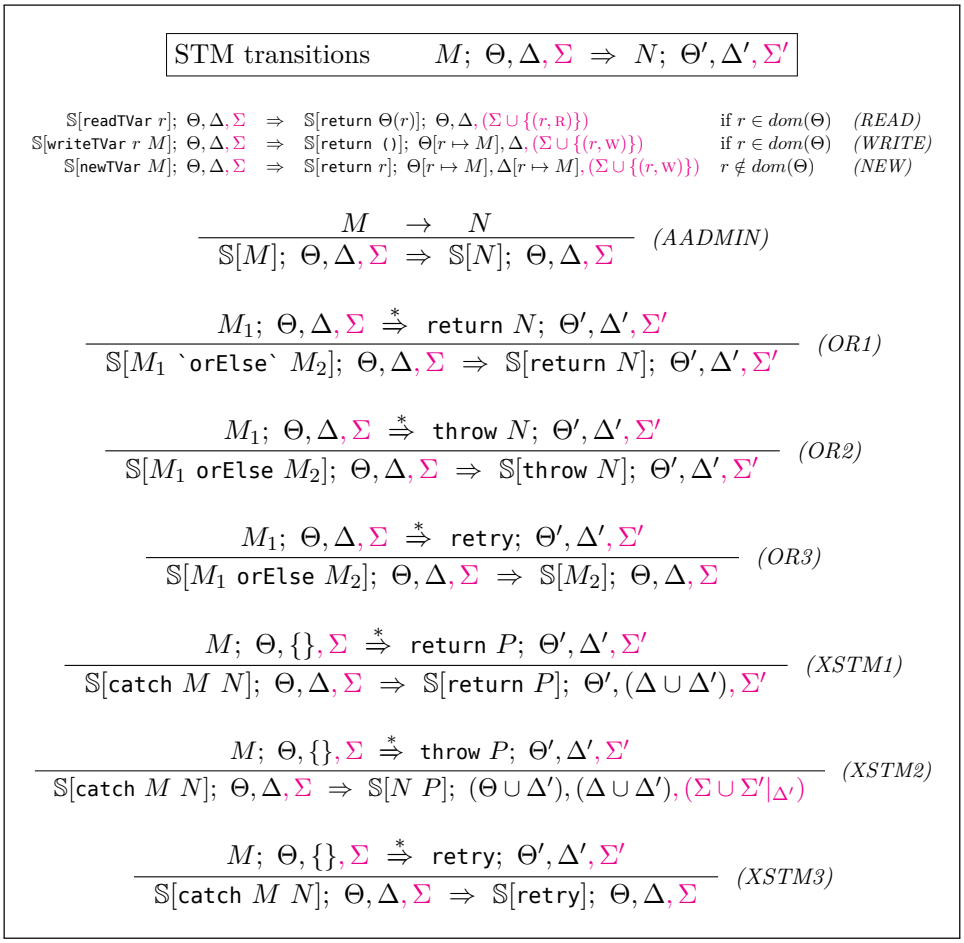

Figure 2.6: STM-level transitions of STM Haskell with finalizers, including nesting checks

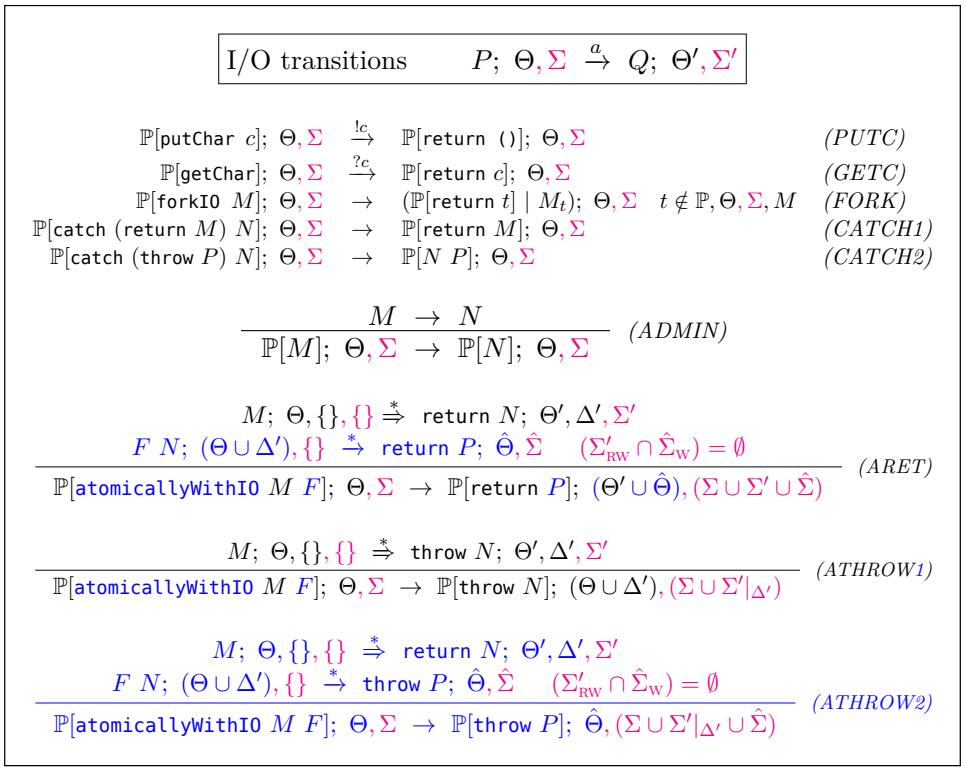

Figure 2.7: I/O-level transitions of STM Haskell with finalizers, including nesting checks

to, respectively. As a shorthand notation, we define $\Sigma_{\mathrm{W}} = \{r \mid (r, \mathrm{W}) \in \Sigma\}$, i.e. the set of references in Σ that have been used in a write operation. Σ_{R} and Σ_{RW} are defined analogously. We also define $\Sigma|_{\Delta} = \{(r, \mathrm{W}) \in \Sigma \mid r \in dom(\Delta)\}$, which restricts Σ to those labelled references that correspond to the allocations recorded in Δ.

Secondly, the STM transition rules *(READ)*, *(WRITE)* and *(NEW)* modify Σ by recording their respective operations. All other rules treat Σ in very much the same way as the heap.

Finally, the purpose of Σ becomes clear on the I/O-level, where rules *(ARET)* and *(ATHROW2)* include a new condition above the line: The set of memory locations read from or written to during the STM transition (Σ'_{RW}) and the set of memory locations written to during the finalizing I/O transition ($\hat{\Sigma}_{\mathrm{W}}$) must be entirely distinct. There must not be a single shared reference between the two phases – except for references that the finalizer has only read. This is exactly the restriction necessary to avoid deadlocks.

2.5.2 Adding invariants

GHC's STM implementation includes the `alwaysSucceeds` primitive, used to introduce dynamically-checked data invariants. It was first described in Harris and Peyton Jones (2006) (where it was called `check`). I have ignored invariants thus far since they increase the overall complexity of the semantics and are mostly orthogonal to finalizers. It is now time to introduce them. Figures 2.8 to 2.10 constitute the complete and final syntax and semantics of STM Haskell with finalizers.

Firstly, we add the set of invariants Ω as another piece of state associated with all I/O and STM transitions. It is usually treated like Σ or the heap.

Secondly, the new STM transition rules *(CHECK1)* and *(CHECK2)* describe the `alwaysSucceeds` primitive. If the invariant holds at the point it is proposed, *(CHECK1)* adds it to Ω and discards its evaluation effects. *(CHECK2)* applies when the invariant throws an exception. As usual, the exception is propagated, the invariant discarded and only allocation effects are retained.

Finally, the biggest changes happen on the I/O level: *(ARET)* becomes *(ARET1)*; along with *(ATHROW2)* it has a new premise that evaluates the invariants in place at the end of the STM transition. When evaluating each invariant, its return value and all its heap effects are discarded. Only if all invariants evaluate to `return` terms can we run the finalizer of `atomicallyWithIO`. Note that it is required that the write set of the I/O transition ($\hat{\Sigma}_{\mathrm{W}}$) is distinct from all of the memory locations referenced by the invariants ($\bigcup \Sigma'_i$).

If any of the invariants throws an exception, the new rule *(ARET2)* applies. Similar to *(ATHROW1)*, the exception is propagated to the I/O level and only allocation effects are preserved (including the allocation effects

	x, y	$\in$	$Variable$
	r, t	$\in$	$Name$
	c	$\in$	$Char$
Value	V	$::=$	$r \mid c \mid$ `\` x `->` M
		$\mid$	`return` $M \mid M$ `>>=` N
		$\mid$	`putChar` $c \mid$ `getChar`
		$\mid$	`throw` $M \mid$ `catch` $M\ N$
		$\mid$	`retry` $\mid M$ `` `orElse` `` N
		$\mid$	`newTVar`
		$\mid$	`readTVar` $r \mid$ `writeTVar` $r\ M$
		$\mid$	`forkIO` M
		$\mid$	`alwaysSucceeds` M
		$\mid$	`atomicallyWithIO` $M\ X$
Term	M, N	$::=$	$x \mid V \mid M\ N \mid \ldots$
Thread soup	P, Q	$::=$	$M_t \mid (P \mid Q)$
Heap	Θ	$::=$	$r \hookrightarrow M$
Allocations	Δ	$::=$	$r \hookrightarrow M$
Touched references	Σ	$::=$	$r \times (\mathrm{R} \mid \mathrm{W})$
Invariants	Ω	$::=$	$\{M\}$
Evaluation contexts	$\mathbb{S}$	$::=$	$[\cdot] \mid \mathbb{S}$ `>>=` M
	$\mathbb{P}$	$::=$	$\mathbb{S}_t \mid (\mathbb{P} \mid P) \mid (P \mid \mathbb{P})$
Action	a	$::=$	$!c \mid ?c \mid \epsilon$

Figure 2.8: The syntax of STM Haskell with invariants and finalizers, including nesting checks

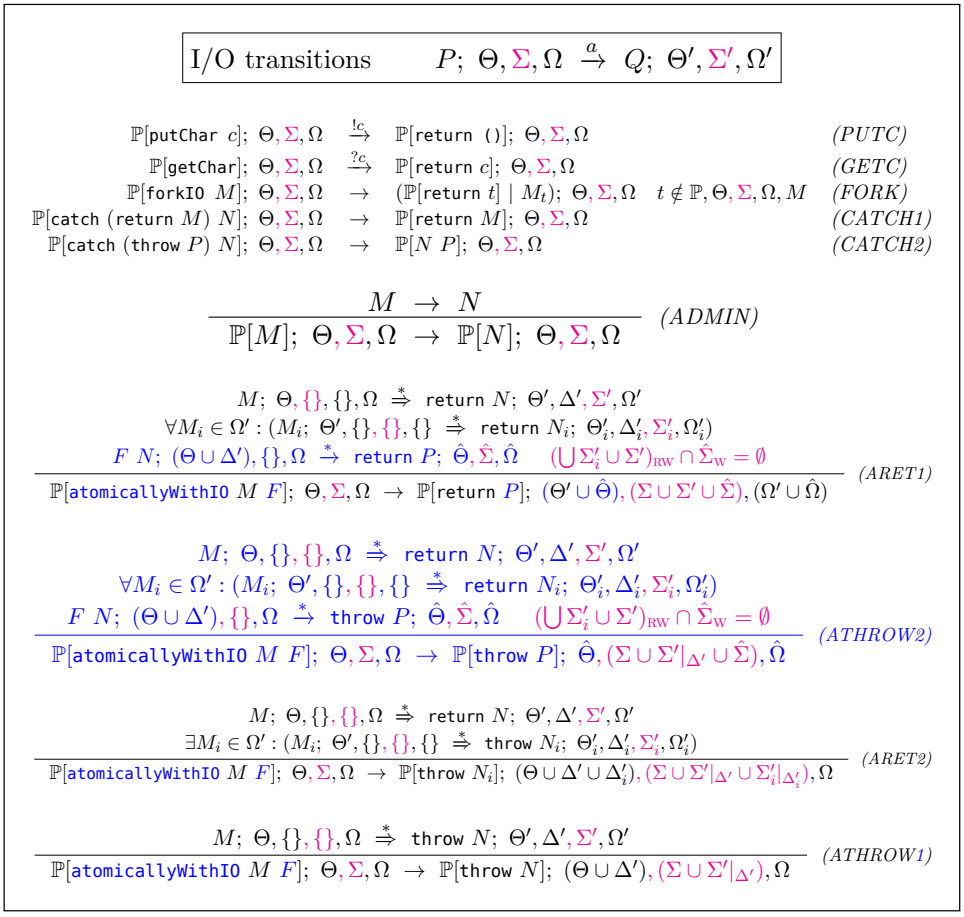

Figure 2.9: I/O-level transitions of STM Haskell with invariants and finalizers, including nesting checks

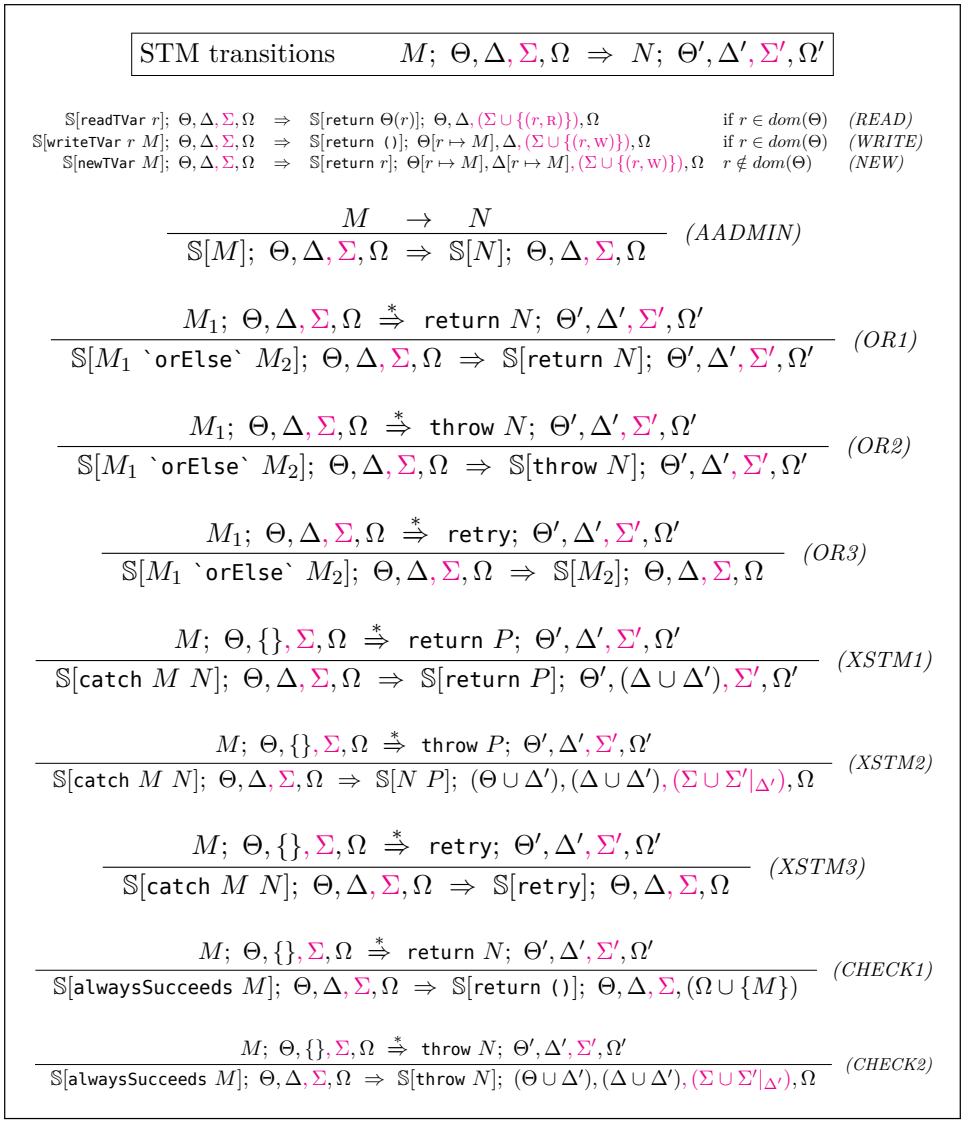

Figure 2.10: STM-level transitions of STM Haskell with invariants and finalizers, including nesting checks

Δ_i' of the broken invariant M_i). The I/O action is not executed in this case.

2.6 Implementation

The changes necessary to add finalizers to GHC's STM implementation are surprisingly few. Before describing them I will once again first give a brief overview of the status quo. To avoid getting bogged down in minutiae, my description of the existing implementation will be a careful simplification, focusing only on the relevant parts of the system. The interested reader is referred to the STM Commentary (Yates 2013) for a more thorough description, and to the GHC source code itself for all the gory details.[4] A fork of GHC containing the changes described in this section is available at `http://github.com/mcschroeder/ghc-stm-finalizers`.

2.6.1 Original STM interface

When evaluating the `atomically` function, the runtime system pushes an `ATOMICALLY_FRAME` onto the execution stack of the current thread. It then calls `stmStartTransaction` to initialize a new thread-local *transactional record* (`TRec`). During the transaction, all read and write operations are recorded in the `TRec`. Every `TVar` accessed by the transaction has an entry in the record. Each entry contains a reference to the value of the `TVar` at the start of the transaction (`expected_value`) and a reference to the new value the `TVar` will have when the transaction commits (`new_value`). If a `TVar` has only been read during a transaction, these two values will be identical. The `TVar`s themselves are not modified until the transaction commits.

When execution returns to the `ATOMICALLY_FRAME`, the commit is initiated by calling `stmCommitTransaction`. First, the `TRec` is *validated* by checking if the `expected_value` of each entry is pointer-equal to the `current_value` field of the corresponding `TVar`, i.e. if the `TVar` has changed in between the transaction starting and committing. If everything is as expected, the `TVar` is locked by setting `current_value` to point to the committing `TRec`. If there is a mismatch, i.e. the `TVar` was modified by someone else, the `TRec` is discarded and the transaction restarted. When validation is successful for the whole `TRec`, the `TVar`s are one by one updated by setting their `current_value` to the `new_value` of the corresponding `TRec` entry, which also unlocks them again.

Because all `TVar`s are locked during validation and each one is only unlocked after it has been updated, the whole commit happens atomically with respect to other transactions. Any other transaction trying to commit at the same time will fail its validation. A transaction trying to read a locked `TVar` will briefly block, but since `TVar`s are only ever locked for very short periods of time, this is not a big deal.

[4] `http://git.haskell.org/ghc.git`

```
// Basic transaction execution
TRec *stmStartTransaction();
Closure *stmReadTVar(TRec *trec, TVar *tvar);
void stmWriteTVar(TRec *trec, TVar *tvar, Closure *new_value);

// Transaction commit operations
Bool stmPrepareToCommitTransaction(TRec *trec);
void stmCommitTransaction(TRec *trec);

// Blocking operations
Bool stmWait(TRec *trec);
Bool stmReWait(TRec *trec);

// Transactional variable
struct TVar {
  Closure    *current_value;
  TRec       *frozen_by;
}

// Transactional record entry
struct TRecEntry {
  TVar        *tvar;
  Closure     *expected_value;
  Closure     *new_value
  TRecEntry   *next_entry;
}

// Transactional record
struct TRec {
  TRec         *enclosing_trec;
  TRecEntry    *next_entry;
}
```

Figure 2.11: Simplified STM runtime interface and data types, extended to support finalizers

2.6.2 Adding atomicallyWithIO

The finalizer given to atomicallyWithIO should be executed after it is guaranteed that the transaction can no longer abort due to conflicts with other transactions, but before the transaction's memory effects are made visible. This is only possible at one moment: during `stmCommitTransaction`, between the successful validation of the `TRec` and the updating of the `TVar`s, while all the `TVar`s are still locked.

To achieve this, the validation part of `stmCommitTransaction` has been split off into the new function `stmPrepareToCommitTransaction`. Committing is now a three-part sequence: first, `stmPrepareToCommitTransaction` is called to validate the transactions; if successful, a new `ATOMICALLY_FRAME` is pushed onto the stack and execution jumps to the finalizer code; when the finalizer is done and execution has returned to the `ATOMICALLY_FRAME`, `stmCommitTransaction` is called to finish the commit and update the `TVar`s.

There is one problem: if the `TVar`s are locked while the finalizer is running, then any transaction attempting to access any of those locked `TVar`s will immediately block until the finalizer is finished. This is obviously bad for performance, especially considering that the blocking occurs even when both transactions are read-only. Giving up on read-parallelism is clearly unacceptable. But not locking the `TVar`s while the finalizer is running is also not possible: the transaction could then unknowingly become invalidated by the actions of another transaction and still commit, resulting in an inconsistent state.

The solution: we *freeze* the `TVar`s during finalization. This puts them into a kind of in-between state, where they can still be read by other transactions, but any transaction that attempts to commit an update to a frozen `TVar` will block until the `TVar` is unfrozen again. The new `TVar` field `frozen_by` is set during `stmPrepareToCommitTransaction`, after all `TVar`s have been successfully validated, and points to the finalizing `TRec`; `frozen_by` is cleared again during `stmCommitTransaction`. Although the `TVar`s need to be locked briefly when modifying their `frozen_by` fields (or indeed any other of their fields), they no longer need to be locked for the whole duration of the finalizer.

Unfortunately, it is necessary to lock *all* `TVar`s that are involved in a transaction, *if* the transaction does indeed have a finalizer. Usually a read-phase is used, meaning `TVar`s that are only read but not updated within the atomic block do not get locked at all, not even briefly. This is merely a performance optimization. It would be possible to do it in the presence of finalizers, but it complicates the implementation, since the `frozen_by` field is protected by the `TVar` lock. Note that this read-phase is also disabled in the original implementation whenever a transaction touches an invariant, for the very same reason.

What exactly happens when a transaction tries to commit an update but hits upon a frozen `TVar`? The transaction could simply restart, like it does

whenever the `current_value` of a `TVar` changes behind its back. However, if the `TVar` is frozen, that means it is involved in a potentially long-running finalizer, so it could still be frozen when it is time to commit again. Since the transaction can never make progress as long as the `TVar` is frozen, the only logical solution is to wait until the state of the `TVar` has actually changed.[5] This is similar to what happens when calling the blocking operator `retry`. In fact, we can reuse a lot of `retry`'s runtime machinery: if `stmPrepareToCommitTransaction` discovers that some of the transaction's `TVars` are frozen, it calls `stmWait` to put the `TRec` on the `TVars`' watch queues before putting the thread to sleep. Being on the `TVars`' watch queues means that whenever one of those `TVars` is updated in a commit, the waiting `TRec` is woken up. When this happens, `stmReWait` is called to validate the `TRec` again to see if it should continue waiting or if it makes sense now to restart the transaction or even to commit it right away.

2.6.3 Nesting

The original STM code was written under the assumption that nested transactions are impossible. To ensure safety, any attempts to bypass the type system are caught at run-time. For example, making calls of the following form will result in an error:

```
atomically (unsafeIOToSTM (atomically...))
```

However, the system does implicitly support two other kinds of nesting: (1) the `orElse` operator begins a new `TRec` for each of its branches; if a branch completes successfully (i.e. doesn't retry) the `TRec` is merged upwards into the enclosing record; (2) invariants introduced by `alwaysSucceeds` are checked at commit time, where a new nested transaction is created for each invariant and completely discarded once the check is over.

The form of nesting made possible by `atomicallyWithIO`, viz. beginning a new transaction within the finalizer of another transaction, is similar in execution to these two, except that we want to neither merge nor discard the nested `TRec`, but simply commit it as-is and then return to the old transaction. The only difference between a regular atomic block and an atomic block begun inside a finalizer is that in the latter case the thread has a dormant `TRec` it returns to once the transaction inside the finalizer has finished. Supporting this is pretty easy, as the existing infrastructure for nested `TRecs`, mainly the `enclosing_trec` field, can be reused. Supporting nested transactions basically amounts to loosening some assertions about what constitutes an outermost transaction in a chain of `TRecs`: previously,

[5] Additionally, there is an issue with GHC's optimizer being too eager when removing yield points. The transaction would just restart over and over again, denying the thread running the finalizer the chance to actually complete. This is a long-standing problem in GHC. See http://ghc.haskell.org/trac/ghc/ticket/367.

the outermost transaction was indicated by an empty `enclosing_trec` field, while now it can also be a transaction whose `enclosing_trec` is running a finalizer. Everything else works exactly as it did before.

What happens when the nested transactions share variables? The inner transaction just reading from a `TVar` is never a problem. It simply sees the `current_value` of the `TVar` in memory. Note that since the outer transaction has not yet committed, any updates it might have made to this `TVar` are still only in its `TRec`. They are not yet globally visible, including within the transactions own finalizer. This matches the semantics. As does the fact that an inner transaction can not write to a shared `TVar`. If it tries to, it will block during commit, since the `TVar` has been frozen by the outer transaction. Meanwhile, the outer transaction is waiting for the inner transaction to finish. The runtime system immediately detects this deadlock and throws an exception, gracefully aborting the whole transaction.

Chapter 3

STM as a database language

TODO: this chapter is still in very rough shape

I now present a reusable framework for constructing application-specific databases, built on STM with finalizers. This is a continuation of my previous work on the `tx` library (Schröder 2013), which was the original motivation for finalizers.

What is an "application-specific" database? Fundamentally, such a database is little more than a collection of regular Haskell data types, using pure Haskell as its query language. There is no external data format and no type conversions are necessary. By lifting the familiar functions from the underlying STM abstraction, data can be manipulated directly. There is little separation between functions over the database and the application's business logic. The database layer is very thin.

Durability is enabled by a write-ahead log recording all operations on the database, which otherwise exists solely in-memory. A similar scheme has been implemented by the `acid-state` library (Himmelstrup 2014), which uses a custom lock-based state container instead of deferring to STM. The fact that `acid-state` is used by Hackage[1], the official Haskell package repository, demonstrates the practicality of such a database design.

All code snippets from this and subsequent chapters are taken directly from fully working prototypes. The complete sample code is available at `https://github.com/mcschroeder/stmfin-examples`.

3.1 Example: a social network

As a motivating example, we will build a simple social networking site. The full application, which includes a Haskell web server that exposes a RESTful API and a simple JavaScript client, can be found in the directory `social1`

[1] `http://hackage.haskell.org`

in the sample code. Here, we will focus mainly on the site's back-end, i.e. its database and business logic.

Our social network will start out with a modest set of features:

- Users can post messages to their *timelines*.
- Users can *follow* other users.
- Each user has a personalized *feed*, which interweaves her timeline with the timelines of all the people she follows.

Let's look at some types:

```
data SocialDB = SocialDB
  {users :: TVar (Map UserName User)
  , posts :: TVar (Map PostId Post)
  }
data User = User
  {name      :: UserName
  , timeline  :: TVar [Post]
  , following :: TVar (Set User)
  , followers :: TVar (Set User)
  }
data Post = Post
  {postId :: PostId
  , author :: User
  , time   :: UTCTime
  , body   :: Text
  }
type UserName = Text
newtype PostId = PostId Word64
```

Users are identified by their name and are represented by the User type. Each user keeps her own list of the posts on her timeline and also keeps track of other users that she follows and that are following her. Posts are identified by a globally unique PostId and are represented by the Post type, which contains the body of the post as well as its author and the time it was created. The whole of the social network is contained in the SocialDB type.

Since our types contain transactional variables, computations must be done in the STM monad. For example, this is how we compute a user's feed:

```
feed :: User → STM [Post]
feed user = do
  myPosts ← readTVar (timeline user)
  others ← Set.toList <$> readTVar (following user)
```

```
    otherPosts ← concat <$> mapM (readTVar ∘ timeline) others
    return $ sortBy (flip $ comparing time) (myPosts ++ otherPosts)
```

By using STM, we can write high-level code and get atomicity for free. Additionally, we can take advantage of some other nice STM features, like composable blocking:

```
waitForFeed :: User → UTCTime → STM [Post]
waitForFeed user lastSeen = do
  let isNew post = diffUTCTime (time post) lastSeen > 0.1
  posts ← takeWhile isNew <$> feed user
  if null posts then retry else return posts
```

The retry operator enables waitForFeed to block until there are new posts in a user's feed. Our server uses this function to effortlessly implement HTTP long polling (Loreto et al. 2011).

3.2 The TX monad

If we want to do more than just query the database, we need to go beyond the standard STM monad. To perform durable updates, we use a monad called TX, which is built on top of STM and lets us record database operations. The idea is not to serialize the data itself, but rather to log the function calls that are responsible for updating the data, so that later on they can be replayed to restore the last consistent state of the database.

The following example demonstrates the two main things we do in TX: lifting functions from the underlying STM monad using liftSTM and recording database operations using record. Note how TX is parameterized by the type of database, in our case SocialDB:

```
follow :: User → User → TX SocialDB ()
follow user1 user2 = do
  record $ Follow (name user1) (name user2)
  liftSTM $ do
    modifyTVar (following user1) (Set.insert user2)
    modifyTVar (followers user2) (Set.insert user1)
```

Because it is not possible to simply serialize functions, record takes a type denoting a particular operation:

```
record :: Operation d → TX d ()
```

where Operation d is an associated type (Chakravarty et al. 2005) of the Database class:

```
class Database d where
  data Operation d
  replay :: Operation d → TX d ()
```

In the above example, the follow function is denoted by the Follow constructor of the Operation SocialDB type, which is declared as part of the Database instance for SocialDB:

```
instance Database SocialDB where
  data Operation SocialDB = ... | Follow UserName UserName | ...
  replay (Follow name1 name2) = do
    user1 ← getUser name1
    user2 ← getUser name2
    user1 `follow` user2
```

A database is simply any type that declares operations that can be replayed in the TX monad. Any additional constraints on the Operation d type, e.g. how it is to be transformed into bits for serialization, are up to the storage back-end, which will be discussed in a short while. For now, you may have noticed how the Follow constructor takes as arguments the names of the users, instead of the users themselves. This is because the User data type is not serializable, regardless of any possible constraints put forth by the storage back-end, since it contains TVars. Thus we have to fetch the users again during replay.

But instead of recording these high-level operations, why not simply record every writeTVar? We may not be able to serialize a TVar but we could serialize the *contents* of a TVar. Could we not have a function

```
durableWriteTVar :: Serializable a ⇒ TVar a → a → TX d ()
```

that automatically logs updates at the lowest level, removing the need for explicit calls to record? Alas, this is not possible: during replay, there is no way to automatically re-associate the recorded value with the TVar it originally belonged to. We always need some kind of context to find the one place in our data structure that has the correct TVar to put the value back into. By recording only high-level operations, we get this context for free.

We also get greater flexibility in how and when to record operations. With greater flexibility comes greater responsibility, however: not only does the programmer have to remember to actually record a specific operation, she must be especially careful when composing functions that record something. Consider the following situation:

```
f = record F ≫ g
g = record G
```

Running f first records F and then calls g and so also records G. Replaying this recording will first replay F, which runs f which calls g, and then it will replay G, which calls g again. We've now called g twice, even though it was only executed once during the original run!

That such things are possible is unfortunate, but it keeps the design simple. In appendix TODO I present a different design, based on effectful monads, that aims to eliminate these trade-offs using advanced type system features.

Before we go on to the implementation of TX, let us briefly look at how it handles failure. For example, what happens if getUser can not find a name in the database? Usually, we would want to make possible failure explicit via types, and so we would expect getUser to have a type like

```
getUser :: Text → TX SocialDB (Maybe User)
```

However, consider the following scenario:

```
do eve ← newUser "Eve"
   adam ← getUser "Adam"
   case adam of
     Just adam → eve `follow` adam
     Nothing   → return ()
```

If getUser returns Nothing, then even though the block returns without further action, *eve* is still created, since newUser was called at the very beginning. Of course this particular example is somewhat contrived, as we could easily rearrange the functions to postpone the creation of *eve*. But you can imagine that this may not always be possible and does not scale well.

The real problem is that there is no way to explicitly abort an STM transaction. This is why getUser is actually implemented like this:

```
getUser :: UserName → TX SocialDB User
getUser name = do
  db ← getData
  usermap ← liftSTM $ readTVar (users db)
  case Map.lookup name usermap of
    Just user → return user
    Nothing   → throwTX (UserNotFound name)
```

where throwTX is simply

```
throwTX :: Exception e ⇒ e → TX d a
throwTX = liftSTM ∘ throwSTM
```

We are using exceptions, instead of encoding failure into the types, because exceptions are the only way to properly abort an STM transaction. In

the majority of uses of a function like getUser, the desired outcome in case the user is not found is to abort the transaction. There are certainly ways to avoid those spooky exceptions as much as possible, such as making TX an exception/error/failure monad. But in the end an exception would have to be thrown somewhere regardless. I have again favored the simpler design and it seems to work well in practice.

3.3 Implementation

Internally, the TX monad is a combination reader/writer monad. It has a pretty straightforward implementation:[2]

```
data TX d a = TX {unTX :: d → STM (a, [Operation d])}
instance Functor (TX d) where
  fmap f m = TX $ λd → do (a, ops) ← unTX m d
                          return (f a, ops)
instance Applicative (TX d) where
  pure  = return
  (<*>) = ap
instance Monad (TX d) where
  return a = TX $ λ_ → return (a, [])
  m >>= k  = TX $ λd → do (a, ops) ← unTX m d
                          (b, ops') ← unTX (k a) d
                          return (b, ops ++ ops')
record :: Operation d → TX d ()
record op = TX $ λ_ → return ((), [op])
liftSTM :: STM a → TX d a
liftSTM m = TX $ λ_ → m >>= λa → return (a, [])
getData :: TX d d
getData = TX $ λd → return (d, [])
```

The reader part of the monad, viz. keeping around the database d, is not necessary for the main purpose of recording database operations. It is however very convenient to have a function like getData, allowing the programmer easy access to the root database type from within any TX action.

Now, how is a TX computation actually performed? The TX equivalent for atomically is runTX, and this is where finalizers finally come into play:

```
runTX :: DatabaseHandle s d → TX d a → IO a
runTX h m = atomicallyWithIO action finalizer
```

[2] To be found in the directory `tx1` in the sample code.

```
  where
    action            = unTX m (database h)
    finalizer (a, ops) = serializer h ops ≫ return a
```

The first argument to runTX is a DatabaseHandle, which is an abstraction over the concrete storage and serialization mechanism chosen by the programmer:

```
data DatabaseHandle s d = DatabaseHandle
  { database :: d
  , storage  :: s
  , serializer :: [Operation d] → IO ()
  }
```

For the social network example, I implemented binary serialization to an append-only log file, using the `cereal` (Kolmodin, Lemmih, and Dijk 2013) and `safecopy` (Himmelstrup and Lessa 2014) libraries. The user of this particular storage backend can obtain a DatabaseHandle by calling a function named openDatabase, which deserializes and replays a previously recorded database log, if available, and then prepares the log file for further serialization. Its implementation is given below[3]. Note how the SafeCopy class constraint on Operation is local to this storage module; Database and the TX operations can stay completely agnostic as to how serialization is actually done.

```
data LogFile = LogFile {fileHandle :: MVar Handle}
openDatabase :: (Database d, SafeCopy (Operation d))
             ⇒ FilePath → d → IO (DatabaseHandle LogFile d)
openDatabase fp d = do
  fileHandle ← mkFileHandle fp
  deserialize fileHandle d
  return DatabaseHandle {database = d
                         , storage  = LogFile fileHandle
                         , serializer = serialize fileHandle
                         }
```

Since the LogFile back-end is built on SafeCopy, it supports schema migration right out of the box. Look in the `social2` directory of the sample code for a version of our social network that has some additional features, such as users being able to like posts and to post messages to the timelines of other users. It is a practical example of how to extend and change data types and business logic while keeping compatibility with old database files.

[3] The full module can be found in the sample code under `tx1/TX/LogFile.hs`

Chapter 4

Transactional Tries

The fundamental data type of STM is the transactional variable. A TVar stores arbitrary data, to be accessed and modified in a thread-safe manner. For example, I might define a bank account as

```
type Euro = Int
type Account = TVar Euro
```

and then use a function like

```
transfer :: Account → Account → Euro → STM ()
```

to safely – in the transactional sense – move money between accounts.

But where do those accounts come from? If I am a bank, how do I represent the whole collection of accounts I manage, in a way that is transactionally safe? The obvious solution, and a common pattern, is to simply use an existing container type and put that type into a TVar:

```
type IBAN = String
type Bank = TVar (Map IBAN Account)
```

Since looking up an account from the Map involves a readTVar operation, the Map is entangled with the transaction, and I can be sure that when transferring money between accounts, both accounts actually exist in the bank at the time when the transaction commits.

The drawback of this pattern of simply wrapping a Map inside a TVar is that when adding or removing elements of the Map, one has to replace the Map inside the TVar wholesale. Thus all concurrently running transactions that have accessed the Map become invalid and will have to restart once they try to commit. Depending on the exact access patterns, this can be a serious cause of contention. For example, one benchmark running on a 16-core machine, with 16 threads each trying to commit a slice out of 200 000

randomly generated transactions, resulted in over 1.4 millions retries[1]. That is some serious overhead!

The underlying problem is that the whole Map is made transactional, when we only ever care about the subset of the Map that is relevant to the current transaction. If transaction A updates an element with key k_1 and transaction B deletes an element with key k_2, then those two transactions only really conflict if $k_1 = k_2$; if $k1$ and $k2$ are different, then there is no reason for either of the transactions to wait for the other one. But the Map does not know it is part of a transaction, and the TVar does not know nor care about the structure of its contents. And so the transactional net is cast too wide.

The solution is to not simply put a Map, or any other ready-made container type, into a TVar, but to design data structures specifically tailored to the needs of transactional concurrency. In this chapter, I present one such data structure, the *transactional trie*, based on the concurrent trie of Prokopec, Bagwell, and Odersky (2011). I will describe its design, show major parts of its implementation in Haskell and discuss the trade-offs that have to be made in the transactional setting. I will then evaluate it against similar STM-specialized data structures.

The finalizers of Chapter 2 were a macro-level approach to add side-effects to STM, connecting whole transactions with potentially large I/O actions, tailored to the use case of serializing data. The transactional trie incorporates side-effects on the micro-level. It exposes a transactionally safe interface, while internally circumventing some transactional safety measures in a controlled and manually verified way.

4.1 Background

The transactional trie is based on the concurrent trie of Prokopec, Bagwell, and Odersky (2011), which is a non-blocking concurrent version of the hash array mapped trie first described by Bagwell (2001).

A hash array mapped trie is a tree whose leaves store key-value bindings and whose nodes are implemented as arrays. Each array has 2^k elements. To look up a key, you take the initial k bits of the key's hash as an index into the root array. If the element at that index is another array node, you continue by using the next k bits of the hash as an index into that second array. If that element is another array, you again use the next k bits of the hash, and so on. Generally speaking, to index into an array node at level l, you use the k bits of the hash beginning at position $k * l$. This procedure is repeated until either a leaf node is found or one of the array nodes does not have an entry at the particular index, in which case the key is not yet

[1] A more detailed breakdown of this benchmark can be found in Section 4.4

present in the trie. The expected depth of the trie is $O(log_{2^k}(n))$, which means operations have a nice worst-case logarithmic performance.

Most of the array nodes would only be sparsely populated. To not waste space, the arrays are actually used in conjunction with a bitmap of length 2^k that encodes which positions in the array are actually filled. If a bit is set in the bitmap, then the (logical) array contains an element at the corresponding index. The actual array only has a size equal to the bit count of the bitmap, and after obtaining a (logical) array index i in the manner described above, it has to be converted to an index into the sparse array via the formula $\#((i-1) \wedge bmp)$, where $\#$ is a function that counts the number of bits and bmp is the array's bitmap. To ensure that the bitmap can be efficiently represented, k is usually chosen so that 2^k equals the size of the native machine word, e.g. on 64-bit systems $k = 6$.

The *concurrent* trie extends the hash trie by adding *indirection nodes* above every array node. An indirection node simply points to the array node underneath it. Indirection nodes have the property that they stay in the trie even if the nodes above or below them change. When inserting an element into the trie, instead of directly modifying an array node, an updated copy of the array node is created and an atomic compare-and-swap operation on the indirection node is used to switch out the old array node for the new one. If the compare-and-swap operation fails, meaning another thread has already modified the array while we weren't looking, the operation is retried from the beginning. This simple scheme, were indirection nodes act as barriers for concurrent modification, ensures that there are no lost updates or race conditions of any kind, while keeping all operations completely lock-free. A more thorough discussion, including proofs of linearizability and lock-freedom, can be found in the paper by Prokopec et al. (2011). A Haskell implementation of the concurrent trie, as a mutable data structure in IO, is also available (Schröder 2014).

The *transactional* trie is an attempt to lift the concurrent trie into an STM context. The idea is to use the lock-freedom of the concurrent trie to make a non-contentious data structure for STM. This is not entirely straightforward, as there is a natural tension between the atomic compare-and-swap operations of the concurrent trie, which are pessimistic and require execution inside the IO monad, and optimistic transactions as implemented by STM. While it is possible to simulate compare-and-swap using TVars and retry[2], this would entangle the indirection nodes with the rest of the

[2]Like this, for example:

```
stmCAS :: TVar a → a → a → STM ()
stmCAS var old new = do
  cur ← readTVar var
  if (cur ≡ old)
    then writeTVar var new
    else retry
```

transaction, which is exactly the opposite of what we want. To keep the non-blocking nature of the concurrent trie, the indirection nodes need to be kept independent of the transaction as a whole, which should only hinge on the actual values stored in the trie's leaves. If two transactions were to cross paths at some indirection node, but otherwise concern independent elements of the trie, then neither transaction should have to retry or block. Side-effecting compare-and-swap operations that run within but independently of a transaction are the only way to achieve this. Alas, the type system, with good reason, won't just allow us to mix IO and STM actions, so we have to circumvent it from time to time using unsafeIOToSTM. We will need to justify every single use of unsafeIOToSTM and ensure it does not lead to violations of correctness. Still, bypassing the type system is usually a bad sign, and indeed we will see that correctness can only be preserved at the cost of memory efficiency, at least in an STM implementation without finalizers.

4.2 Implementation

The version of the transactional trie discussed in this chapter is available on Hackage at `http://hackage.haskell.org/ttrie-0.1`. The full source code can also be found at `http://github.com/mcschroeder/ttrie`.

The module Control.Concurrent.STM.Map[3] exports the transactional trie under the following interface:

```
data Map k v
empty :: STM (Map k v)
insert :: (Eq k, Hashable k) ⇒ k → v → Map k v → STM ()
lookup :: (Eq k, Hashable k) ⇒ k → Map k v → STM (Maybe v)
delete :: (Eq k, Hashable k) ⇒ k → Map k v → STM ()
```

Now let us implement it. As always, we begin with some types:[4]

```
newtype Map k v = Map (INode k v)
type INode k v = IORef (Node k v)
data Node k v = Array !(SparseArray (Branch k v))
              | List  ![Leaf k v]
data Branch k v = I !(INode k v)
                | L !(Leaf k v)
data Leaf k v = Leaf !k !(TVar (Maybe v))
```

[3]The name of the trie's public data type is Map, instead of, say, TTrie. The more general name is in keeping with other container libraries and serves to decouple the interface from the specific implementation based on concurrent tries.

[4]The ! operator is a strictness annotation.

The transactional trie largely follows the construction of a concurrent trie:

- The INode is the indirection node described in the previous section and is simply an IORef, which is a mutable variable in IO. To read and write IORefs atomically, we will use some functions and types from the `atomic-primops` package (Newton 2014):

  ```
  data Ticket a
  readForCAS :: IORef a → Ticket a
  peekTicket :: Ticket a → IO a
  casIORef :: IORef a → Ticket a → a → IO (Bool, Ticket a)
  ```

  The idea of the Ticket type is to encapsulate proof that a thread has observed a specific value of an IORef. Due to compiler optimizations, it would not be safe to just use pointer equality to compare values directly.

- A Node is either an Array of Branches or a List of Leafs. The List is used in case of hash collisions. A couple of convenience functions help us manipulate such collision lists:

  ```
  listLookup :: Eq k ⇒ k → [Leaf k v] → Maybe (TVar (Maybe v))
  listDelete :: Eq k ⇒ k → [Leaf k v] → [Leaf k v]
  ```

  Their implementations are entirely standard.

  The Array is actually a SparseArray, which abstracts away all the bit-fiddling necessary for navigating the bit-mapped arrays underlying a hash array mapped trie. Its interface is largely self-explanatory:

  ```
  data SparseArray a
  emptyArray :: SparseArray a
  mkSingleton :: Level → Hash → a → SparseArray a
  mkPair :: Level → Hash → a → a → Maybe (SparseArray a)
  arrayLookup :: Level → Hash → SparseArray a → Maybe a
  arrayInsert :: Level → Hash → a → SparseArray a → SparseArray a
  arrayUpdate :: Level → Hash → a → SparseArray a → SparseArray a
  ```

  I will not go into the implementation of SparseArray. It is fairly low-level and can be found in the internal Data.SparseArray module of the `ttrie` package.

  Some additional functions are used to manipulate Hashes and Levels. Again, they are self-explanatory:

  ```
  type Hash = Word
  hash :: Hashable a ⇒ a → Hash
  ```

```
type Level = Int
down :: Level → Level
up :: Level → Level
lastLevel :: Level
```

- A Branch either adds another level to the trie by being an INode or it is simply a single Leaf.

The one big difference to a concurrent trie lies in the definition of the Leaf. Basically, a Leaf is a key-value mapping. It stores a key k and a value v. But the way it stores v determines how the trie behaves in a transactional context. Let us build to it step by step:

1. Imagine if Leaf were defined exactly like in a concurrent trie:

   ```
   data Leaf k v = Leaf !k v
   ```

   Then an atomic compare-and-swap on an INode to insert a new Leaf would obviously not be safe during an STM transaction: other transactions could see the new value v before our transaction commits; and they could replace v by inserting a new Leaf for the same key, resulting in our insert being lost.

2. We can eliminate lost inserts by wrapping the value in a TVar:

   ```
   data Leaf k v = Leaf !k !(TVar v)
   ```

   Now, instead of replacing the whole Leaf to update v, we can use writeTVar to only modify the value part of the Leaf. If two transactions try to update the same Leaf, then STM will detect the conflict and one of the transactions would have to retry.

   Of course, if there does not yet exist a Leaf for a specific key, then a new Leaf will still have to be inserted with a compare-and-swap. In this case it is again possible for other transactions to read the TVar immediately after the swap, even though our transaction has not yet committed and may still abort. This can happen without conflict because the new Leaf contains a newly allocated TVar and allocation effects are allowed to escape transactions by design (see Section 2.4). Reading a newly allocated TVar will never cause a conflict.

3. To ensure proper isolation, the actual type of Leaf looks like this:

   ```
   data Leaf k v = Leaf !k !(TVar (Maybe v))
   ```

   By adding the Maybe, we can allocate new TVars with Nothing in them. A transaction can then insert a new Leaf containing Nothing

using the compare-and-swap operation. Other threads will still able to read the new TVar immediately after the compare-and-swap, but all they will get is Nothing. The transaction, meanwhile, can simply writeTVar (Just v) to safely insert the actual value into the Leaf's TVar. If another transaction also writes to the TVar and commits before us, then we have a legitimate conflict on the value level, and our transaction will simply retry.

Now that we have the types that make up the trie's internal structure, we can implement its operations. We begin with the function to create an empty trie:

```
empty :: STM (Map k v)
empty = unsafeIOToSTM $ Map <$> newIORef (Array emptyArray)
```

It contains no surprises, although it has the first use of unsafeIOToSTM, which in this case is clearly harmless.

For the rest of the operations, let us assume we have a function

```
getTVar :: (Eq k, Hashable k) ⇒ k → Map k v → STM (TVar (Maybe v))
```

that either returns the TVar stored in the Leaf for a given key, or allocates a new TVar for that key and inserts it appropriately into the trie. The TVar returned by getTVar k m will always either contain Just v, where v is the value associated with the key k in the trie m, or Nothing, if k is not actually present in m. Additionally, getTVar obeys the following invariants:

Invariant 1: getTVar k_1 $m \equiv$ getTVar k_2 m $\iff$ $k_1 \equiv k_2$

Invariant 2: getTVar itself does not read from nor write to any TVars.

Now we can define the trie's operations as follows:

```
insert k v m = do var ← getTVar k m
                  writeTVar var (Just v)
lookup k m = do var ← getTVar k m
                readTVar var
delete k m = do var ← getTVar k m
                writeTVar var Nothing
```

The nice thing about defining the operations this way, is that correctness and non-contentiousness follow directly from the invariants of getTVar. The first invariant ensures correctness. If we get the same TVar every time we call getTVar with the same key, and if that TVar is unique to that key, then STM will take care of the rest. And if, by the second invariant, getTVar

does not touch any transactional variables, then the only way one of the operations can cause a conflict is if it actually operates at the same time on the same TVar as another transaction. Unnecessary contention is therefore not possible.

All that is left to do is implementing getTVar. Essentially, getTVar is a combination of the insert and lookup functions of the concurrent trie, just lifted into STM. It tries to look up the TVar associated with a given key, and if that does not exist, allocates and inserts a new TVar for that key. When inserting a new TVar, the structure of the trie has to be changed to accommodate the new element.

Let us look at the code:

```
getTVar k (Map root) = go root 0
  where
    h = hash k
```

The actual work is done by the recursive helper function go. It begins at level 0 by looking into the *root* indirection node. Note that throughout the iterations of go, the hash h of the key is only computed once.

```
go inode level = do
  ticket ← unsafeIOToSTM $ readForCAS inode
  case peekTicket ticket of
    Array a → case arrayLookup level h a of
      Just (I inode2) → go inode2 (down level)
      Just (L leaf2@(Leaf k2 var))
        | k ≡ k2        → return var
        | otherwise     → cas inode ticket (growTrie level a (hash k2) leaf2)
      Nothing           → cas inode ticket (insertLeaf level a)
    List xs  → case listLookup k xs of
      Just var          → return var
      Nothing           → cas inode ticket (return ∘ List ∘ (:xs))
```

The use of unsafeIOToSTM here is clearly safe – all we are doing is reading the value of the indirection node. This does not have any side effects, so it does not matter if the transaction aborts prematurely. If the transaction retries, the indirection node is just read again – possibly resulting in a different value. It is also possible that the value of the indirection node changes during the runtime of the rest of the function – but that is precisely why we obtain a Ticket.

Depending on the contents of the indirection node, we either go deeper into the trie with a recursive call of go; return the TVar associated with the key; or insert a new TVar by using the cas function to swap out the old contents of the indirection node with an updated version that somehow contains the new TVar.

The cas function is also part of the **where** clause of getTVar:

```
cas inode ticket f = do
  var ← newTVar Nothing
  node ← f (Leaf k var)
  (ok, _) ← unsafeIOToSTM $ casIORef inode ticket node
  if ok then return var
        else go root 0
```

It implements a transactionally safe compare-and-swap procedure:

1. Allocate a new TVar containing Nothing.
2. Use the given function f to produce a *node* containing a Leaf with this TVar.
3. Use casIORef to compare-and-swap the old contents of the *inode* with the new *node*.
4. If the compare-and-swap was successful, the new *node* is immediately visible to all other threads. Return the TVar to the caller, who is now free to use writeTVar to fill in the final value.
5. If the compare-and-swap failed, because some other thread has changed the *inode* since the time we first read it, restart the operation – not with the STM retry, which would restart the whole transaction, but simply by calling go *root* 0 again.

All that is remaining now are the functions given for f in the code of go. Given a new Leaf, they are supposed to return a Node that somehow contains this new Leaf. In the case of the overflow list, this is just a trivial anonymous function that prepends the leaf into the List node. The insertLeaf function does pretty much the same, except for Array nodes:

```
insertLeaf level a leaf = do
  let a' = arrayInsert level h (L leaf) a
  return (Array a')
```

In case of a key collision, things are a tiny bit more involved. The growTrie function puts the colliding leaves into a new level of the trie, where they hopefully won't collide anymore:

```
growTrie level a h2 leaf2 leaf1 = do
  inode2 ← unsafeIOToSTM $ combineLeaves (down level) h leaf1 h2 leaf2
  let a' = arrayUpdate level h (I inode2) a
  return (Array a')
combineLeaves level h1 leaf1 h2 leaf2
```

```
    | level ⩾ lastLevel = newIORef (List [leaf1, leaf2])
    | otherwise =
      case mkPair level h (L leaf1) h2 (L leaf2) of
        Just pair → newIORef (Array pair)
        Nothing → do
          inode ← combineLeaves (down level) h1 leaf1 h2 leaf2
          let a = mkSingleton level h (I inode)
          newIORef (Array a)
```

The use of casIORef here is once again harmless, as combineLeaves only uses IO to allocate new IORefs. The mkPair function for making a two-element SparseArray curiously returns a Maybe, because it is possible that on a given level of the trie the two keys hash to the same array index and so the leaves can't both be put into a single array. In that case, another new indirection node has to be introduced into the trie and the procedure repeated. If at some point the last level has been reached, the leaves just go into an overflow List node.

4.3 Memory efficiency

While the transactional trie successfully carries over the lock-freedom of the concurrent trie and keeps the asymptotic performance of its operations, it does have to make a couple of concessions regarding memory efficiency.

The first concession is that when looking up any key for the first time, the lookup operation will actually grow the trie. This is a direct consequence of using getTVar to implement the trie's basic operations. If getTVar does not find the Leaf for a given key, it allocates a new one and inserts it. One might wonder if it is possible to implement a lookup function that does not rely on getTVar. The following attempt is pretty straightforward and appears to be correct at first glance – although you might already guess from its name that something is not quite right:

```
phantomLookup :: (Eq k, Hashable k) ⇒ k → Map k v → STM (Maybe v)
phantomLookup k (Map root) = go root 0
  where
    h = hash k
    go inode level = do
      node ← unsafeIOToSTM $ readIORef inode
      case node of
        Array a → case arrayLookup level h a of
          Just (I inode2) → go inode2 (down level)
          Just (L (Leaf k2 var))
            | k ≡ k2      → readTVar var
            | otherwise   → return Nothing
```

```
    Nothing      → return Nothing
List xs  → case listLookup k xs of
    Just var     → readTVar var
    Nothing      → return Nothing
```

The problem with this simple implementation is that under certain circumstances it allows for *phantom reads*. Consider the following pair of functions:

```
f = atomically $ do v1 ← phantomLookup k
                    v2 ← phantomLookup k
                    return (v1 ≡ v2)
g = atomically (insert k 23)
```

Due to STM's isolation guarantees, one would reasonably expect that f always returns True. However, sometimes f will return False when g is run between the two phantomLookups in f. How is this possible? If you start out with an empty trie, then the first phantomLookup in f obviously returns Nothing. And it does so without touching any TVars, because there is no TVar for k at this point. Only when running g for the first time, will a TVar for k be created. The transaction inside f will now happily read from this TVar during the second phantomLookup and will not detect any inconsistencies, because this is the first time it has seen the TVar. This problem does not only occur on an empty trie, but any time we look up a key that has not previously been inserted. The only remedy is to ensure that there is always a TVar for every key, even if it is filled with Nothing, which is exactly what the implementation of lookup using getTVar does.

Granted, it seems as if these kinds of phantom lookups might not occur regularly in practice, and even if they did, they would probably cause no great harm. The overhead of always allocating a Leaf for every key that is ever looked up, on the other hand, seems much more troublesome. However, phantomLookup exhibits exactly the kind of seldom-occurring unexpected behavior that results in bugs that are incredibly hard to find. And having a lookup function that grows the trie is really only an issue in two cases:

1. when we expect the keys we look up to not be present a significant amount of the time; then a transactional trie is probably really not the right data structure. Although if one were to use phantomLookup instead of lookup, and if in this particular scenario phantom lookups are actually acceptable, then using a transactional trie could still be feasible.

2. when a malicious actor purposefully wants to increase memory consumption, i.e. a classic denial-of-service attack; then one can again counteract this by using phantomLookup, limited to those places that

are susceptible to attack. For example, a login routine in a web application could first use phantomLookup to check if the user actually exists, before continuing with the transaction. Here the phantom lookup does not matter, because if the user does not exist the transaction is aborted anyway.

Thus, it makes sense to have the behavior of lookup be the default and provide phantomLookup for those select scenarios where it is actually an improvement.

The other trade-off the trie has to make regarding memory efficiency, is that the delete operation does not actually remove Leafs or compact the trie again. It merely fills a Leaf's TVar with Nothing. This frees up the values associated with the keys, which is the major part of a trie's memory consumption, but it does not delete the keys or compress the structure that has emerged in the trie, which might now be suboptimal given the trie's current utilization.

Again, for the common use case, this might not be a problem. Very often, we do not want to actually delete certain data, but merely mark it as deleted; or maybe delete the data, but mark the associated keys as having been previously in use in order to prevent reusing them. Think of unique user IDs, for example. In such a scenario, the overhead of the trie not actually deleting Leafs disappears. Still, there are of course cases where we do want the trie to always be as compact a representation of its data as possible, and there is in fact a way to achieve this: by using finalizers. What prevents us from just removing Leafs from the trie during a transaction, is that transactions might get restarted or aborted. If we use the normal delete function during the transaction, which essentially just marks a key as deleted (but by accessing the TVar ensures that there are no conflicts with other transactions), we can then use an unsafeDelete function inside the transaction's finalizer to really remove the Leaf:

```
atomicallyWithIO (delete k m) (\_ → unsafeDelete k m)
```

The finalizer ensures the atomicity of the otherwise unsafe operation.

To implement unsafeDelete properly and preserve lock-freedom, we have to slightly alter the Node type by adding an additional kind of node: a Tomb node.

```
data Node k v = Array  !(SparseArray (Branch k v))
              | List   ![Leaf k v]
              | Tomb !(Leaf k v)
```

A Tomb node holds a single key. It comes into existence when unsafeDelete would result in an Array node with only a single Leaf beneath it. A Tomb

node is the last value assigned to an INode. If any operation encounters an INode that points to a Tomb node, it cannot modify the INode but rather must help compress the trie by merging the tombed leaf into the parent INode before retrying the operation. The exact details of unsafeDelete and the accompanying cleanup and compression procedures are very tricky, but not especially interesting. They are described in great detail by Prokopec, Bronson, et al. (2012) and are included in the final implementation of the transactional trie.

4.4 Evaluation

I empirically evaluated the transactional trie against similar data structures, measuring contention, runtime performance and memory allocation. The benchmarks were run on an Amazon EC2 C3 extra-large instance with Intel Xeon E5-2680 v2 (Ivy Bridge) processors and a total of 16 physical cores. Under comparison were a transactional trie; a HashMap from the `unordered-containers` library (Tibell and Yang 2014), wrapped inside a TVar; and the STM-specialized hash array mapped trie from the `stm-containers` library (Volkov 2014b).

Each benchmark performs a number of random STM transactions involving one of the container types. The benchmarks differ in size and composition of the transactions. In all cases, the keys used for operations on the containers are random Text strings and the values simply (), except for the `unordered-containers` case, were the exact type of container is TVar (HashMap Text (TVar ())). The benchmarks are run multiple times, using an increasing number of threads. The transactions are split evenly over the number of threads in use. The time it takes to complete all transactions for a particular container is measured using the `criterion` and `criterion-plus` libraries (O'Sullivan 2014; Volkov 2014a), which calculate the mean execution time over many iterations. To measure contention, the transactions are run again using the `stm-stats` library (Leuschner, Wehr, and Breitner 2011) to count how often the STM runtime system had to restart transactions due to conflicts. Finally, the transactions are run once more to measure the total amount of allocated memory, using GHCs built-in facilities for collecting memory usage statistics. All benchmarks were compiled using GHC 7.8.3. For more details about test data generation and the finer points of the benchmark setup, see the `ttrie` source distribution.

The first four benchmarks (Figure 4.1) each perform 200 000 transactions, where each transaction is just a single operation: insert, update, lookup or delete. The insertion benchmark starts out with an empty container, while all other benchmarks operate on containers prefilled with 200 000 entries. An update operation is simply an insert where the key is already present in the container.

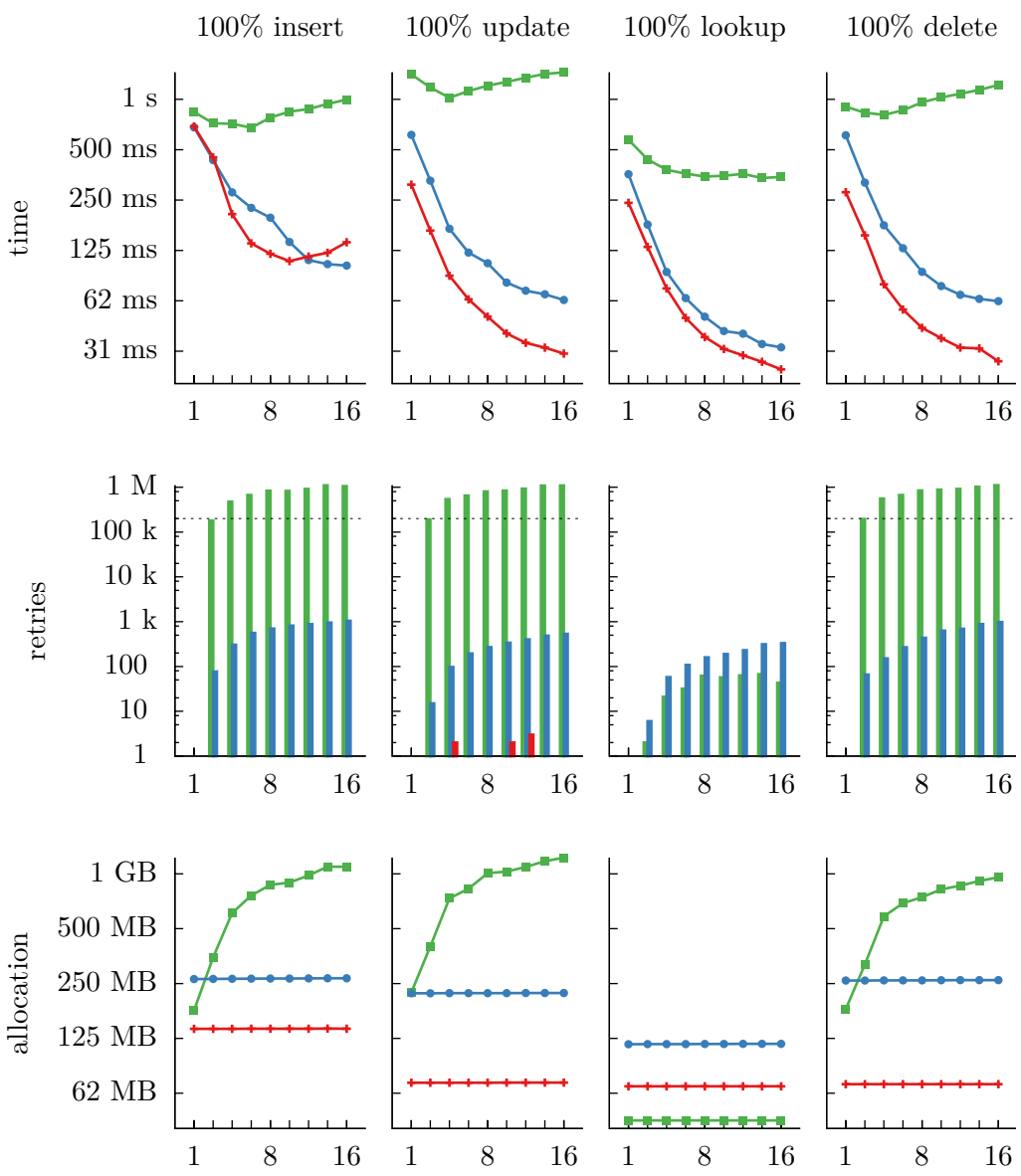

Figure 4.1: Single operation benchmarks:
200 000 transactions of size 1;
under comparison: **ttrie**, **stm-containers**, **unordered-containers**

Unsurprisingly, the TVar-wrapped HashMap from `unordered-containers` is much slower across the board than the STM-specialized data structures. The pattern of wrapping a container in a single TVar does not scale well at all. In fact, any kind of potential performance gain through increased concurrency is vastly overshadowed by contention. As soon as the number of retries exceeds the number of transactions, i.e. every transaction has to retry at least once, additional threads are actually detrimental to performance. With 16 threads, each one performing only 12 500 transactions, the insert, update and delete benchmarks recorded over 1 million retries. This kind of contention not only increases a transaction's run time but also its memory footprint. While the total amount of memory allocated by `ttrie` and `stm-containers` stays constant irrespective of the level of concurrency, for `unordered-containers` memory consumption increases dramatically with the number of threads.

The transactional trie, as expected, exhibits no contention at all. The few retries recorded during the update benchmark are due to legitimate conflicts. It handily beats `unordered-containers` in every benchmark, many times by an order of magnitude, except for memory allocation during lookup. It is twice as fast as `stm-containers` during update and delete, and about 30% faster during lookup; it has consistently better memory performance.

For the second set of benchmarks (Figure 4.2), transactions are no longer just a single operation, but a mix of up to 5 operations. The insert benchmark now consists of 70% inserts, 10% updates, 10% lookups and 10% deletes; the update benchmark of 10% inserts, 70% updates, 10% lookups and 10% deletes; the lookup benchmark of 10% inserts, 10% updates, /0% lookups and 10% deletes; and the delete benchmark of 10% inserts, 10% updates, 10% lookups and 70% deletes. The insert benchmark again starts out with an empty container, while the other benchmarks operate on containers now prefilled with 1 000 000 entries. Using transactions of varying sizes and with different compositions much closer reflects real-world usage.

As expected, legitimate conflicts between transactions are now slightly more common, evidenced by the increased number of retries measured for the transactional trie, but they still amount to less than a dozen in the worst case. In comparison, `unordered-containers` causes more than 1.3 million retries in the worst case. By incorporating, to varying degrees, all operation types in each benchmark, the different benchmarks are now much closer in results than before. The edges are smoothed out, so to speak. Consequently, the transactional trie now always outperforms the other container types, on both time and memory, and with a greater lead. The sole exception is the insert benchmark, where runtime performance is virtually identical to that of `stm-containers`.

The last benchmark (Figure 4.3) again measures 200 000 transactions,

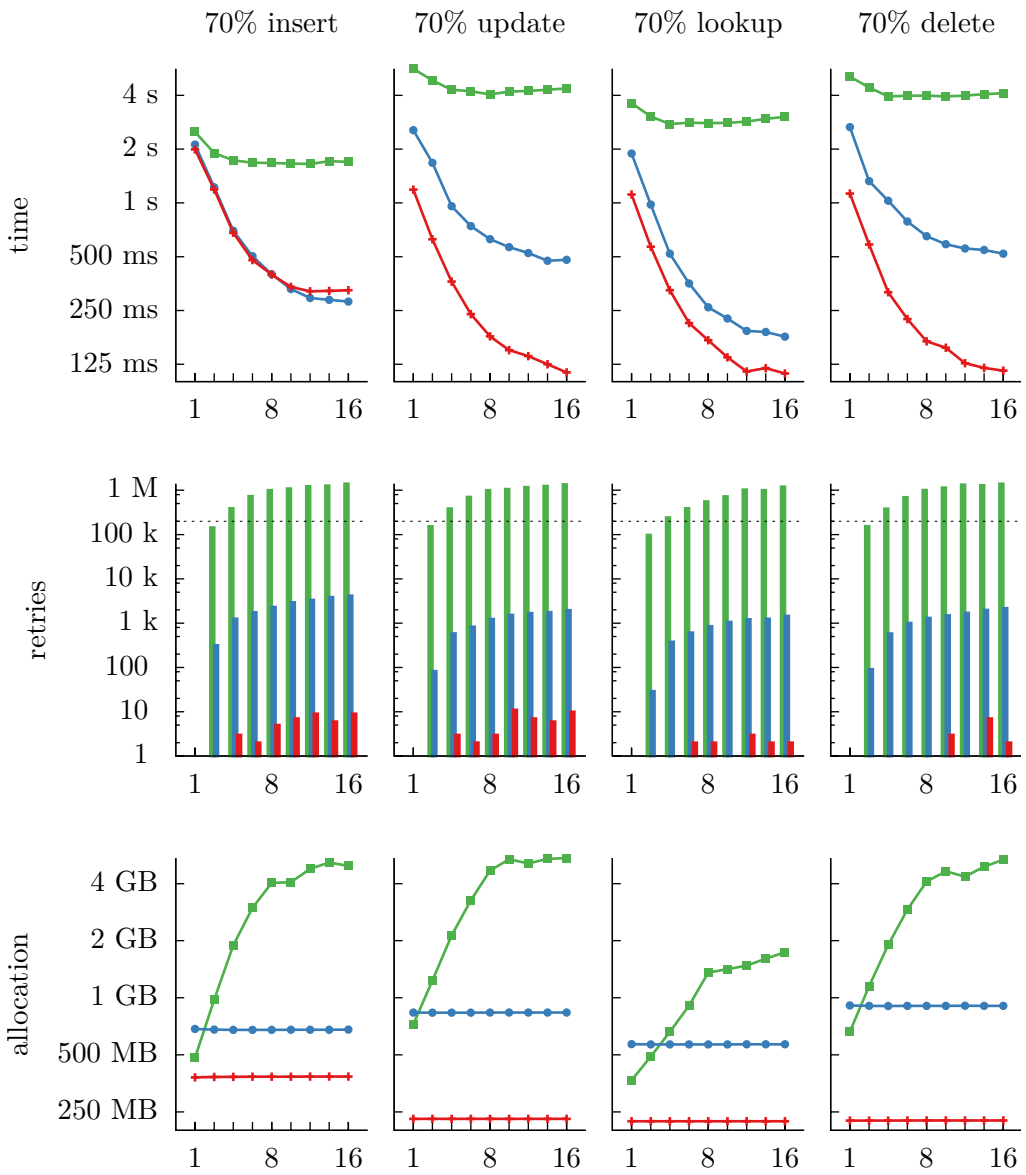

Figure 4.2: Benchmarks with 200 000 transactions of size 1-5; under comparison: **ttrie**, **stm-containers**, **unordered-containers**

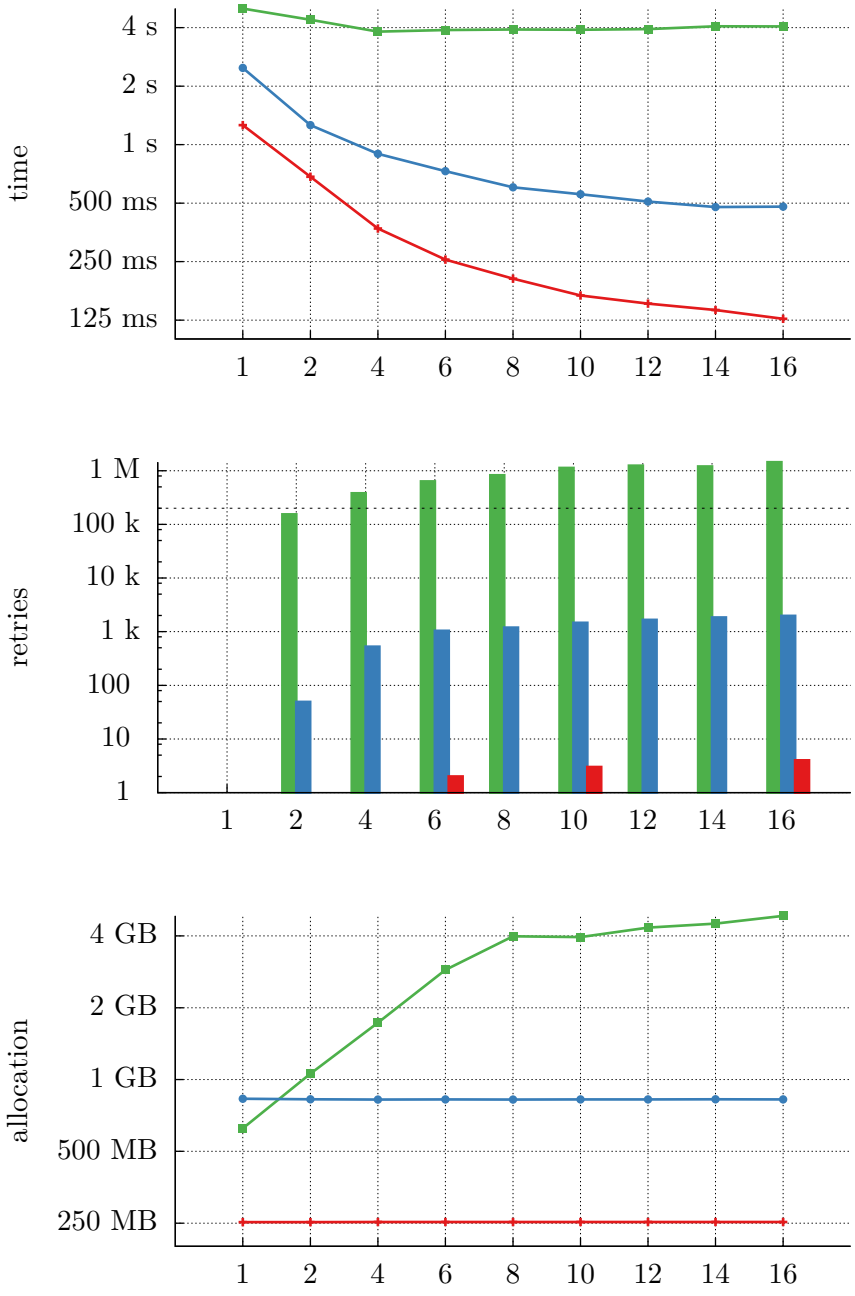

Figure 4.3: Balanced benchmark: 200 000 transactions of size 1-5;
25% inserts, 25% updates, 25% lookups, 25% deletes;
container prefilled with 1 000 000 keys;
under comparison: **ttrie**, **stm-containers**, **unordered-containers**

each up to 5 operations long, but now with a balanced mix of 25% inserts, 25% updates, 25% lookups and 25% deletes, on containers prefilled with 1 000 000 entries.

This kind of benchmark plays to the strengths of the transactional trie: it is 2–4 times faster than `stm-containers`, allocating only a third of the memory; and 4–30 times faster than `unordered-containers`, allocating almost 20 times less memory.

Chapter 5

Conclusions & Perspectives

Bibliography

Bagwell, Phil (2001). *Ideal Hash Trees*. Tech. rep. LAMP-REPORT-2001-001. EPFL (cit. on p. 40).

Chakravarty, Manuel M. T. et al. (2005). "Associated Types with Class". In: Principles of Programming Languages. (2005). POPL. ACM, pp. 1–13 (cit. on p. 33).

Harris, Tim, Simon Marlow, et al. (2005). "Composable Memory Transactions". In: Principles and Practice of Parallel Programming. (2005). PPoPP. ACM, pp. 48–60 (cit. on pp. 3, 11–14, 16, 17).

Harris, Tim and Simon Peyton Jones (2006). "Transactional memory with data invariants". In: Languages, Compilers, and Hardware Support for Transactional Computing. (2006). TRANSACT. ACM (cit. on pp. 3, 11, 21).

Himmelstrup, David (2014). *The `acid-state` package*. Version 0.12.2. URL: http://hackage.haskell.org/package/acid-state-0.12.2 (cit. on p. 31).

Himmelstrup, David and Felipe Lessa (2014). *The `safecopy` package*. Version 0.8.3. URL: http://hackage.haskell.org/package/safecopy-0.8.3 (cit. on p. 37).

Kolmodin, Lennart, Lemmih, and Bas van Dijk (2013). *The `cereal` package*. Version 0.4.0.1. URL: http://hackage.haskell.org/package/cereal-0.4.0.1 (cit. on p. 37).

Leuschner, David, Stefan Wehr, and Joachim Breitner (2011). *The `stm-stats` package*. Version 0.2.0.0. URL: http://hackage.haskell.org/package/stm-stats-0.2.0.0 (cit. on p. 51).

Loreto, Salvatore et al. (2011). *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. RFC 6202 (cit. on p. 33).

Marlow, Simon (2013a). *Parallel and Concurrent Programming in Haskell*. O'Reilly Media Inc. (cit. on p. 3).

— (2013b). *The `async` package*. Version 2.0.1.5. URL: http://hackage.haskell.org/package/async-2.0.1.5 (cit. on p. 9).

Newton, Ryan (2014). *The `atomic-primops` package*. Version 0.6. URL: http://hackage.haskell.org/package/atomic-primops-0.6 (cit. on p. 43).

O'Sullivan, Bryan (2014). *The `criterion` package*. Version 1.0.2.0. URL: http://hackage.haskell.org/package/criterion-1.0.2.0 (cit. on p. 51).

Prokopec, Aleksander, Phil Bagwell, and Martin Odersky (2011). *Cache-Aware Lock-Free Concurrent Hash Tries*. Tech. rep. EPFL-REPORT-166908. EPFL (cit. on pp. 40, 41).

Prokopec, Aleksander, Nathan Grasso Bronson, et al. (2012). "Concurrent tries with efficient non-blocking snapshots". In: Principles and Practice of Parallel Programming. (2012). PPoPP. ACM, pp. 151–160 (cit. on p. 51).

Robinson, Peter and Chris Kuklewicz (2012). *The `stm-io-hooks` package*. Version 0.7.5. URL: http://hackage.haskell.org/package/stm-io-hooks-0.7.5 (cit. on p. 11).

Schröder, Michael (2013). *The `tx` package*. Version 0.1.0.0. URL: http://hackage.haskell.org/package/tx-0.1.0.0 (cit. on p. 31).

— (2014). *The `ctrie` package*. Version 0.1.0.2. URL: http://hackage.haskell.org/package/ctrie-0.1.0.2 (cit. on p. 41).

Tibell, Johan and Edward Z. Yang (2014). *The `unordered-containers` package*. Version 0.2.5.0. URL: http://hackage.haskell.org/package/unordered-containers-0.2.5.0 (cit. on p. 51).

Volkov, Nikita (2014a). *`criterion-plus`: Enhancement of the `criterion` benchmarking library*. Version 0.1.3. URL: http://hackage.haskell.org/package/criterion-plus-0.1.3 (cit. on p. 51).

— (2014b). *The `stm-containers` package*. Version 0.2.3. URL: http://hackage.haskell.org/package/stm-containers-0.2.3 (cit. on p. 51).

Yates, Ryan (2013). *GHC Commentary: Software Transactional Memory (STM)*. URL: http://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/STM (cit. on p. 25).