Region-Based Dynamic Separation for STM Haskell [*]

Laura Effinger-Dean
University of Washington
effinger@cs.washington.edu

Dan Grossman
University of Washington
djg@cs.washington.edu

Abstract

We present the first design and implementation of dynamic separation in STM Haskell. Dynamic separation is a recent approach to software transactional memory (STM) that achieves strongly-atomic semantics with performance comparable to that of a weakly-atomic STM. STM Haskell, a lazy-versioning STM library for Haskell, previously supported strongly-atomic semantics via static separation, and we have found dynamic separation to be a natural extension of the library's interface.

Our approach to dynamic separation makes several advances over previous work. First, we use a notion of regions to allow entire data structures to share a protection state, which avoids expensive and unnecessary data-structure traversals. Second, we enrich the set of protection states with a "thread-local" state that allows data to be used inside and outside transactions. Third, we support static and dynamic separation, and in particular use a well-typed interface to allow all dynamic-separation code to be safely composed with static-separation code.

We prove the correctness of region-based dynamic separation with respect to an operational semantics for a lazy-versioning STM using the Coq theorem prover. We have also evaluated the performance of our system on a suite of STM Haskell programs.

1. Introduction

Transactional constructs for high-level languages must negotiate the tension between programmability, efficiency of nontransactional operations, and correctness. Ideally, languages would provide well-defined constructs that preserve the strong safety guarantees that make transactions appealing to programmers.

STM Haskell is an implementation of transactions for the Glasgow Haskell Compiler [10]. The library, which extends Concurrent Haskell, is beautifully designed, allowing transactions to be composed at run time in sequence or as alternatives. Haskell's purely-functional core and monadic type system are surprisingly well-suited to the STM programming model, as the type system prevents conflicts between transactional and nontransactional code.

STM Haskell's division of mutable variables into transactional and nontransactional types neatly sidesteps a tricky issue in STM semantics and implementations: weak vs. strong atomicity. In so-called *weakly-atomic* implementations, nontransactional reads and writes bypass the STM entirely. If such accesses conflict with transactional accesses, they can compromise high-level semantic guarantees such as atomicity and isolation in surprising ways (see Section 2.2 for an example). Yet using the STM mechanism for all memory accesses (to ensure strong atomicity) can be expensive and/or introduce problems with legacy code.

Solutions to this dilemma provide the performance of weak atomicity and the semantics of strong atomicity. The *static separation* of transactional and nontransactional memory in STM Haskell suffices, as has been proven in more formal settings [1, 15], but is conservative. Several common concurrency idioms, such as publication and privatization, require dynamically changing whether a location is used inside or outside transactions.

Dynamic separation [2, 3] addresses this limitation directly by having programmers make explicit calls to change the protection state of memory locations. For example, when a thread publishes a location, it must change the location's protection state to *protected*—that is, accessible only from transactions. While these state changes interact with the STM, they should be relatively rare compared to nontransactional reads and writes. In prior work, each explicit state change applied to just one mutable location. A key innovation in our work is support for *regions*: groups of locations with the same sharing state that can have their state changed with a single, constant-time operation on a special region object.

In general, this paper implements, evaluates, formally defines, and proves correct dynamic separation as an extension to STM Haskell, including support for regions. STM Haskell is an ideal setting for ensuring that dynamic separation is suitable for a well-defined programming language, compiler, and run-time system. Of course, we hope our results will positively influence other settings, just as the original STM Haskell has proven influential.

The more specific contributions of this work are as follows:

- We present the first implementation of dynamic separation in STM Haskell, and more broadly, the first implementation in a lazy-versioning STM. (STM Haskell uses lazy versioning; prior work assumed eager updates.)
- We define an interface that allows static and dynamic separation to exist side-by side within Haskell's type system.
- We introduce regions to hold the protection state for a collection of locations. This change lets mutable structures share a single state among all their objects, saving tedious and expensive data-structure traversals.
- We have a richer set of protection states than prior work. We support a thread-local state for locations that currently cannot be read or written by other threads, and a read-only state for locations that are currently immutable.
- We define a low-level semantics to model a lazy-versioning STM, and prove that it provides strong atomicity for programs that obey dynamic separation.
- We discuss the details of our STM Haskell implementation, and evaluate it on some small programs.

2. Language Design

We first review STM Haskell's design and the semantic problems introduced by weak atomicity. (Readers may need to supplement this necessarily brief overview with a review of the paper introducing STM Haskell [10] or a more general tutorial on Haskell and monads, e.g., [17].) We then give a high-level overview of our extensions to STM Haskell to support dynamic separation.

[*] This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124.

```
-- IORefs (nontransactional variables)
data IORef a
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()

-- TVars (transactional variables)
data STM a
instance Monad STM

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

Figure 1. Concurrent Haskell's interface for `IORef`s and STM.

2.1 STM Haskell

Haskell is a lazy purely-functional language known for its monadic type system. Although its core is purely functional, the full language includes a complete I/O subsystem. In particular, Haskell supports threading and runs on shared-memory multiprocessors [9]. Threads in Concurrent Haskell communicate via shared memory. One shared-memory mechanism is the `IORef` type, the interface for which is listed in Figure 1.[1] `IORef`s are simple mutable references; if an `IORef` is thread-shared, the programmer must avoid race conditions using synchronization primitives.

Concurrent Haskell supports software memory transactions with STM Haskell, a library with a composable interface for creating modular transactions [10]. STM Haskell's programming interface appears in Figure 1. Although Haskell supports regular mutable references via the `IORef` type, transactions use a special type of reference called a `TVar`, or transactional variable. Transactions generally consist of one or more `TVar` operations sequenced together with monadic bind, like so:

```
increment :: TVar Int -> STM ()
increment t = do { x <- readTVar t;
                   writeTVar t (x + 1) }
```

Note that calling `increment` on a `TVar t` does *not* increment `t`. Rather, it produces an *STM action*, which is only performed when executed with `atomically`.[2]

The advantage of placing STM operations in a separate monad is *composability*: client programmers can combine STM operations exported by different libraries to form larger transactions. This functionality is similar to that provided by nested transactions, but the authors of STM Haskell have carefully thought out the semantics for exceptions and *blocking* (manual transaction abort via `retry`) in relation to composed transactions.

2.2 Semantic Problems with Weak Atomicity

By dividing mutable references into `TVar`s and `IORef`s, which can be accessed only inside or outside transactions, respectively, STM Haskell provides strongly-atomic transaction semantics via static separation. We might naively wish to improve STM Haskell's expressivity by adding weak atomicity to the interface:

[1] A value of type `IO a` is an *I/O action*: a side-effecting computation that, when performed, returns a value of type `a`.

[2] Technically, `atomically` produces an `IO` action, which must in turn be executed by the program's `main` function, but for the purposes of this paper the reader can assume `atomically` executes the transaction.

Thread 1 (T1):	**Thread 2 (T2):**
	`atomically (do {`
`atomically (`	`  sh <- readTVar x_sh;`
`  writeTVar x_sh False);`	`  if sh`
`r1 <- readTVarIO x;`	`    then writeTVar x 1`
`r2 <- readTVarIO x;`	`    else return ()`
	`});`

Initially `x` holds 0 and `x_sh` holds `True`. Can $r1 \neq r2$?

Figure 2. Buffered writes example in STM Haskell.

```
readTVarIO  :: TVar a -> IO a
writeTVarIO :: TVar a -> a -> IO ()
```

`readTVarIO` and `writeTVarIO` read and write `TVar`s without using a transaction. Depending on how they are implemented, these functions may undermine the semantics of transactions.

Consider the program in Figure 2. `x_sh` holds a boolean that indicates whether `x` is shared. Under any reasonable interpretation of the program semantics, it should be the case that T1 sees two identical values for `x`: if T1's transaction runs first, then both reads of `x` yield 0; if T2's transaction runs first, both reads yield 1.

This example exposes *buffered writes*, a known problem in weakly-atomic, lazy-versioning STMs [18]. The critical scenario for Figure 2 is as follows. T2 starts its transaction, reads `True` from `x_sh`, and buffers its update to `x`. The transaction successfully commits, but does not yet write back the new value for `x`. T1 starts its transaction, buffers a write to `x_sh`, and tries to commit. T2 has not finished its write to `x`, but T1's transaction did not access `x`, so T1 successfully commits. T1 completes its transaction by writing back `False` to `x_sh`, then reads 0 from `x`. T2 writes 1 to `x`, and finally T1 reads 1—a different value—from `x`.

This example and others [18] demonstrate that unprotected reads and writes break the high-level semantics of transactions, even for programs that appear to be free of data races. We could solve this problem by having `readTVarIO` and `writeTVarIO` interact with the STM (enforcing strong atomicity), or by not providing these operations (enforcing static separation). This paper focuses on another approach: *dynamic separation*.

2.3 Dynamic Separation

Dynamic separation is a recent approach for providing strongly-atomic semantics with the performance of a weak implementation [2, 3]. In dynamic separation, the programmer explicitly makes a call to the STM whenever a reference changes its "protection state." For example, in Figure 2, `x` goes from being "protected" (shared between multiple threads and accessed only in transactions) to "unprotected" (inaccessible from transactions).

In Figure 2, we can recover strong semantics by calling a new function, `unprotectTVar`, on `x` after T1's transaction completes. This call interacts with the STM, waiting until all transactions currently accessing its argument have completed before continuing. `unprotectTVar` solves the buffered writes problem by forcing T1 to wait for T2's transaction to complete. The threads still coordinate access to `x` using `x_sh`, but `unprotectTVar` eliminates any race conditions caused by the weak implementation.

The discussion thus far considered a hypothetical implementation that allows unprotected access to `TVar`s via `readTVarIO` and `writeTVarIO`. Our actual implementation takes a different approach: we introduce a new type of reference, *dynamic variables*. `DVar`s may be accessed inside or outside a transaction and have a dynamic protection state. Because `TVar`s and `DVar`s are distinct types, programmers can use both dynamic and static separation in STM programs. This seamless combination of static and dynamic separation in a single interface is a novel contribution of our work.

```
data DSTM a
instance Monad DSTM

-- DVars reside in the new DSTM monad
data DVar a
newDVar       :: a -> DSTM (DVar a)
readDVar      :: DVar a -> DSTM a
writeDVar     :: DVar a -> a -> DSTM ()

-- Primitives to change a DVar's protection state
protectDVar   :: DVar a -> IO ()
unprotectDVar :: DVar a -> IO ()
makeRODVar    :: DVar a -> IO ()
makeTLDVar    :: DVar a -> IO ()

-- Functions to allow DSTM code in STM or IO monad
protected     :: DSTM a -> STM a
unprotected   :: DSTM a -> IO a
```

Figure 3. Interface for dynamic separation with per-DVar protection states (extends Figure 1).

Figure 3 extends the interface for STM Haskell with support for dynamic separation. The interface does not include protection regions, which we will add in Section 2.5. Operations on DVars produce results in the DSTM (dynamic STM) monad. DSTM code may be executed in two ways: (1) as part of a transaction, by wrapping a DSTM term with `protected`, or (2) as nontransactional code in the IO monad, by wrapping a DSTM term with `unprotected`. DSTM code may therefore be composed with other transactions in sequence, or as alternatives with `orElse`. If libraries export functions with DSTM return types, then clients can choose whether or not to execute the functions as transactions.

2.4 Read-Only and Thread-Local

The previous formalization of dynamic separation [2] supported two protection states: protected (only accessible in transactions) and unprotected (only accessible outside transactions). The C# implementation [3] also has a read-only state, which allows references to be read by transactional and nontransactional code.

These three states share an important property: if accesses to a DVar comply with its protection state, then it is impossible for a transaction to race with a nontransaction on that DVar. Protected and unprotected DVars may only be accessed inside and outside transactions, respectively, and read-only DVars are clearly safe from data races. This safety property is crucial for providing strong semantics with a weak implementation.

Our system supports protected, unprotected, and read-only states, as well as a *thread-local* state. To be more precise, the thread-local state is a family of related protection states, one for each thread. Thread-local DVars may be read and written inside or outside a transaction, but only by the thread that owns that DVar. The protection state is a policy, not a program invariant: if a transaction attempts to access a DVar that is local to a different thread, the transaction will abort. A thread cannot race with itself, so programs that comply with the protection states of all DVars will continue to have strongly-atomic semantics.

In Figure 3, `makeRODVar` sets a DVar's protection state to be read-only and `makeTLDVar` sets a DVar's protection state to be local to the current thread.

2.5 Region-Based Protection States

Together with the seamless integration of static and dynamic separation, our most important contribution is the addition of protection regions to the interface for dynamic separation. Prior work in C# assumed that each reference's protection state was independent of the protection states of all other references. This per-reference approach is not ideal for shared data structures that consist of many related references. If protection states are per-reference, changing the protection state of a data structure requires iterating through the entire structure, which is tedious and inefficient.

```
data DSTM a
instance Monad DSTM

-- Regions and protection state changes
data DRgn
newDRgn        :: DSTM DRgn
protectDRgn    :: DRgn -> IO ()
unprotectDRgn  :: DRgn -> IO ()
makeRODRgn     :: DRgn -> IO ()
makeTLDRgn     :: DRgn -> IO ()

-- Dynamic variables (now region-allocated)
data DVar a
newDVar        :: a -> DRgn -> DSTM (DVar a)
readDVar       :: DVar a -> DSTM a
writeDVar      :: DVar a -> a -> DSTM ()

-- Functions to allow DSTM code in STM or IO monad
protected      :: DSTM a -> STM a
unprotected    :: DSTM a -> IO a
```

Figure 4. Interface for dynamic separation with region-based protection states (replaces Figure 3).

For example, suppose we implement a binary tree using DVars:

```
data BinaryTreeNode =
    Node { val   :: Int,
           left  :: DVar BinaryTreeNode,
           right :: DVar BinaryTreeNode }
  | Nil
```

To protect all of the DVars in the tree, we would have to traverse the entire tree, protecting each DVar one at a time. This code is tedious to write and takes time proportional to the number of nodes.

Our solution is to introduce special objects called *protection regions*, or DRgns. A region holds the protection state for a group of DVars. Every DVar is associated with a DRgn. The partition of DVars into DRgns is specified by the programmer at allocation time. For shared data structures such as the binary tree, programmers can create a single region object at initialization, and allocate all of the structure's DVars in that region. The programmer must keep track of the structure's region for use in allocations (e.g., by storing it in a field at the root of the tree). Changing the protection state of an entire data structure is now a simple $O(1)$ operation: calling `protectDRgn` (for example) on the structure's region.

Figure 4 shows the final version of our DSTM interface. We introduce a new datatype, DRgn, and primitives for changing the protection state of a DRgn. `newDVar` now takes the DRgn in which to allocate the DVar in addition to the DVar's initial value. Of course, it is still possible to simulate the interface given in Figure 3 by creating a DRgn that contains a single DVar.

3. Formal Semantics

In this section we give a high-level operational semantics for STM Haskell with region-based dynamic separation. This semantics draws on and extends ideas from several sources, including the semantics for dynamic separation in an eager-update STM [2], the Atoms Family languages [15], and our own work on modeling relaxed memory models [8]. We omit several STM Haskell features, including `retry`, TVars, and exceptions.

$$\begin{array}{rl}
 & \texttt{DVar}\ d;\quad \texttt{DRgn}\ r;\quad \text{Thread ID}\ \theta \\
\text{Action}\ a ::= & \mathsf{pure} \mid \mathsf{fork}(\theta) \mid \mathsf{new}(d, r, M) \mid \mathsf{rgn}(r) \\
 & \mid \mathsf{rd}(d, M) \mid \mathsf{wr}(d, M) \mid \mathsf{mkP}(r) \mid \mathsf{mkU}(r) \\
 & \mid \mathsf{mkRO}(r) \mid \mathsf{mkTL}(r) \mid \mathsf{begin} \mid \mathsf{end} \\
\text{Term}\ M ::= & x \mid \texttt{\textbackslash}x\texttt{->}M \mid M\ M \mid () \mid \texttt{return}\ M \mid M \texttt{ >>= } M \\
 & \mid \texttt{newDRgn} \mid \texttt{newDVar}\ M\ M \mid \texttt{readDVar}\ M \\
 & \mid \texttt{writeDVar}\ M\ M \mid \texttt{forkIO}\ M \mid \texttt{atomically}\ M \\
 & \mid \texttt{protected}\ M \mid \texttt{unprotected}\ M \\
 & \mid \texttt{protectDRgn}\ M \mid \texttt{unprotectDRgn}\ M \\
 & \mid \texttt{makeRODRgn}\ M \mid \texttt{makeTLDRgn}\ M \\
 & \mid d \mid r \mid \theta \mid \texttt{inAtomically}\ M \\
\text{Context}\ \mathbb{M} ::= & [\cdot] \mid \mathbb{M} \texttt{ >>= } M \mid \texttt{inAtomically}\ \mathbb{M} \\
 & \mid \texttt{protected}\ \mathbb{M} \mid \texttt{unprotected}\ \mathbb{M} \\
\text{Protection state}\ p ::= & \mathsf{pr} \mid \mathsf{unpr} \mid \mathsf{ro} \mid \mathsf{tl}(\theta) \\
\text{Transaction state}\ s ::= & \cdot \mid \theta \mid \mathsf{com}(\theta)
\end{array}$$

$$\begin{array}{rlrl}
\text{Effect}\ \sigma & ::= (\theta, a) \mid \epsilon & \text{Optional}\ \hat{M} & ::= \cdot \mid M \\
\text{Program state}\ P & : \theta \Rightarrow M & \text{Tag}\ t & ::= \mathsf{RO} \mid \mathsf{Mod} \\
\text{Store}\ S & : d \Rightarrow (r, M) & \text{Log}\ L & : d \Rightarrow (t, M) \\
\text{Region map}\ R & : r \Rightarrow p & \text{Weak heap}\ H & ::= (s, S, R, L)
\end{array}$$

Figure 5. Program and heap syntax.

Our goal is to prove that a lazy-versioning STM is correct for programs that obey dynamic separation. To this end, we establish the equivalence of two sets of semantics: the Strong semantics, which implements strong atomicity, and the Weak semantics, which models a lazy-versioning STM. We omit the full Strong semantics for space reasons, but it is a restricted version of the Weak semantics.[3] Although these two semantics are not equivalent in general, they are equivalent for programs that obey dynamic separation.

We actually use three semantics to define our system. First, the *program semantics* defines rewrite rules for Haskell terms. Next we have the two aforementioned *heap semantics*, Strong and Weak, which rewrite the global heap. Separating the program and heap semantics means that we can reuse the program syntax and semantics, which is elegant and avoids errors due to redundancy.

3.1 Syntax

The syntax for programs and heaps is given in Figure 5. We assume that there is a map type $k \Rightarrow v$, with empty map $[]$. We add values to a map m with $m[k \mapsto v]$, retrieve values with $m(k)$, and remove values with $m_{/k}$. The domain of a map m is $\mathsf{dom}(m)$.

Common syntax Actions a describe program steps. An effect σ is either a pair of a thread ID and an action, or the empty effect ϵ. The program and heap semantics must agree on the effect for each step. For example, a program step might appear to read a value out of thin air, but the value is supplied by the heap semantics. The empty effect is for heap steps (e.g., committing a log entry) that are invisible to the program.

Program syntax Terms M represent Haskell expressions. In addition to the DSTM interface, we include an `inAtomically` runtime form to distinguish between active and inactive transactions. Evaluation contexts $\mathbb{M}$ identify the active subexpression in a term. The program state P is a finite map from thread IDs to terms.

Heap syntax A Weak heap H consists of a transaction state s, a store S, a region map R, and a transaction log L. The transaction state s is $\cdot$ if no thread is in a transaction, θ if thread θ is in an active transaction, and $\mathsf{com}(\theta)$ if thread θ is committing its transaction. Each `DVar` is allocated in a `DRgn`, so the store S maps `DVar` locations to `DRgn`/term pairs. p represents the protection state for a region. The four possible protection states are pr (protected), unpr (unprotected), ro (read-only) and $\mathsf{tl}(\theta)$ (local to thread θ). The region map R maps `DRgns` to protection states. The transaction log L maps `DVars` to terms, each paired with a tag t that records whether the entry is read-only (RO) or modified (Mod).

3.2 Program Semantics

We present the program semantics as a helper judgment in Figure 6. The pure evaluation judgment ($\rightsquigarrow$) (omitted) represents Haskell's pure core. The single-threaded judgment ($\hookrightarrow$) rewrites terms within monadic contexts, emitting an action a describing the action performed, as well as an optional forked thread $\hat{M}$. The multithreaded judgment ($\rightharpoondown$) transitions the program state P, emitting an action a and the ID θ of the thread that performed the action.

3.3 Weak Semantics

Figure 7 gives the Weak semantics, which is inspired by our implementation. At most one transaction executes at a time, and nontransactional reads and writes (READ and WRITE) may race with transactions. Although real STM implementations, including ours, allow multiple transactions to execute simultaneously, reasoning about one transaction at a time is still interesting because of conflicts with nontransactional code.

BEGIN sets the current transaction state to the transaction's thread ID. Transactions read values into the log (TXREAD1) and modify only the logged copies (TXWRITE). Reads of logged `DVars` always see the logged value (TXREAD2). After the transaction completes (END), it commits all the values in the log, writing back modified entries (COMMITW) and removing read-only entries without updating the store (COMMITR). ENDCOMMIT allows other threads to begin executing transactions once the log is empty.

A key restriction is that protection state change operations (e.g., MAKEP—the other three are analogous and omitted) must occur when no transaction is active. This restriction reflects the intuition that these operations essentially act as barriers that prevent interference between transactions and non-transactions. In the real implementation, we require these operations to lock whatever region they are accessing, rather than waiting for all transactions to complete.

As with the program semantics, each step of the Weak heap semantics emits an effect. The system judgment ($\longrightarrow$) allows both the program and the heap to progress together (MUTUAL), and also allows the heap to take empty steps (HEAP).

3.4 Strong Semantics

The Strong semantics (omitted for space) is a restricted version of the Weak semantics that implements strong atomicity. We change the Weak semantics as follows:

- The heap no longer has a log. Transactional operations read and write the store directly.
- We delete the COMMITR, COMMITW, ENDCOMMIT, and HEAP rules, and have END transition directly to state $\cdot$. Therefore the transaction state $\mathsf{com}(\theta)$ is no longer necessary.
- Nontransactional reads and writes are allowed only when the current transaction state is $\cdot$ (i.e., no transaction is active).[4]

[3] Our unabridged semantics and Coq code are available at `http://wasp.cs.washington.edu/wasp_scat.html`.

[4] The Strong semantics actually combines the transactional and nontransactional rules for reads and writes.

$$\boxed{M \xhookrightarrow{a} M'; \hat{M}} \qquad \textsc{Eval}\ \frac{M \rightsquigarrow M'}{\mathbb{M}[M] \xhookrightarrow{\mathsf{pure}} \mathbb{M}[M']; \cdot} \qquad \boxed{P \xrightarrow{(\theta,a)} P'} \qquad \textsc{PrgmStep}\ \frac{P(\theta) = M \qquad M \xhookrightarrow{a} M'; \cdot}{P \xrightarrow{(\theta,a)} P[\theta \mapsto M']}$$

$$\textsc{PrgmFork}\ \frac{\begin{array}{c} P(\theta) = M \qquad \theta' \notin \mathsf{dom}(P) \\ M \xhookrightarrow{\mathsf{fork}(\theta')} M'; M'' \end{array}}{P \xrightarrow{(\theta,\mathsf{fork}(\theta'))} P[\theta \mapsto M'][\theta' \mapsto M'']}$$

$$\begin{array}{rcl@{\qquad}rcl}
\mathbb{M}[\texttt{newDRgn}] & \xhookrightarrow{\mathsf{rgn}(r)} & \mathbb{M}[\texttt{return}\ r]; \cdot & \mathbb{M}[\texttt{makeTLDRgn}\ r] & \xhookrightarrow{\mathsf{mkTL}(r)} & \mathbb{M}[\texttt{return}\ ()]; \cdot \\
\mathbb{M}[\texttt{newDVar}\ r\ M] & \xhookrightarrow{\mathsf{new}(d,r,M)} & \mathbb{M}[\texttt{return}\ d]; \cdot & \mathbb{M}[\texttt{forkIO}\ M] & \xhookrightarrow{\mathsf{fork}(\theta)} & \mathbb{M}[\texttt{return}\ \theta]; M \\
\mathbb{M}[\texttt{readDVar}\ d] & \xhookrightarrow{\mathsf{rd}(d,M)} & \mathbb{M}[\texttt{return}\ M]; \cdot & \mathbb{M}[\texttt{atomically}\ M] & \xhookrightarrow{\mathsf{begin}} & \mathbb{M}[\texttt{inAtomically}\ M]; \cdot \\
\mathbb{M}[\texttt{writeDVar}\ d\ M] & \xhookrightarrow{\mathsf{wr}(d,M)} & \mathbb{M}[\texttt{return}\ ()]; \cdot & \mathbb{M}[\texttt{inAtomically}\ (\texttt{return}\ M)] & \xhookrightarrow{\mathsf{end}} & \mathbb{M}[\texttt{return}\ M]; \cdot \\
\mathbb{M}[\texttt{protectDRgn}\ r] & \xhookrightarrow{\mathsf{mkP}(r)} & \mathbb{M}[\texttt{return}\ ()]; \cdot & \mathbb{M}[\texttt{return}\ M'\ \texttt{>>=}\ M] & \xhookrightarrow{\mathsf{pure}} & \mathbb{M}[M\ M']; \cdot \\
\mathbb{M}[\texttt{unprotectDRgn}\ r] & \xhookrightarrow{\mathsf{mkU}(r)} & \mathbb{M}[\texttt{return}\ ()]; \cdot & \mathbb{M}[\texttt{protected}\ (\texttt{return}\ M)] & \xhookrightarrow{\mathsf{pure}} & \mathbb{M}[\texttt{return}\ M]; \cdot \\
\mathbb{M}[\texttt{makeRODRgn}\ r] & \xhookrightarrow{\mathsf{mkRO}(r)} & \mathbb{M}[\texttt{return}\ ()]; \cdot & \mathbb{M}[\texttt{unprotected}\ (\texttt{return}\ M)] & \xhookrightarrow{\mathsf{pure}} & \mathbb{M}[\texttt{return}\ M]; \cdot
\end{array}$$

Figure 6. Program semantics.

$$\boxed{\mathsf{active}(s,\theta)} \quad \textsc{NotCommit}\ \frac{s \neq \mathsf{com}(\theta)}{\mathsf{active}(s,\theta)} \quad \boxed{\mathsf{notTx}(s,\theta)} \quad \textsc{NotTxNone}\ \frac{}{\mathsf{notTx}(\cdot,\theta)} \quad \textsc{NotTxOther}\ \frac{\theta \neq \theta'}{\mathsf{notTx}(\theta,\theta')} \quad \textsc{NotTxCommit}\ \frac{\theta \neq \theta'}{\mathsf{notTx}(\mathsf{com}(\theta),\theta')}$$

$$\boxed{H \xrightarrow{\sigma} H'} \quad \textsc{Pure}\ \frac{\mathsf{active}(s,\theta)}{(s,S,R,L) \xrightarrow{(\theta,\mathsf{pure})} (s,S,R,L)} \quad \textsc{NewRgn}\ \frac{\mathsf{active}(s,\theta) \qquad r \notin \mathsf{dom}(R)}{(s,S,R,L) \xrightarrow{(\theta,\mathsf{rgn}(r))} (s,S,R[r \mapsto \mathsf{pr}],L)}$$

$$\textsc{New}\ \frac{\mathsf{active}(s,\theta) \qquad d \notin \mathsf{dom}(S)}{(s,S,R,L) \xrightarrow{(\theta,\mathsf{new}(d,r,M))} (s,S[d \mapsto (r,M)],R,L)} \quad \textsc{Read}\ \frac{\mathsf{notTx}(s,\theta) \qquad S(d) = (r,M)}{(s,S,R,L) \xrightarrow{(\theta,\mathsf{rd}(d,M))} (s,S,R,L)}$$

$$\textsc{Write}\ \frac{\mathsf{notTx}(s,\theta) \qquad S(d) = (r,M)}{(s,S,R,L) \xrightarrow{(\theta,\mathsf{wr}(d,M'))} (s,S[d \mapsto (r,M')],R,L)} \quad \textsc{Fork}\ \frac{\mathsf{notTx}(s,\theta)}{(s,S,R,L) \xrightarrow{(\theta,\mathsf{fork}(\theta'))} (s,S,R,L)} \quad \textsc{MakeP}\ \frac{}{(\cdot,S,R,L) \xrightarrow{(\theta,\mathsf{mkP}(r))} (\cdot,S,R[r \mapsto \mathsf{pr}],L)} \quad \ldots$$

$$\textsc{Begin}\ \frac{}{(\cdot,S,R,L) \xrightarrow{(\theta,\mathsf{begin})} (\theta,S,R,L)} \quad \textsc{End}\ \frac{}{(\theta,S,R,L) \xrightarrow{(\theta,\mathsf{end})} (\mathsf{com}(\theta),S,R,L)}$$

$$\textsc{TxRead1}\ \frac{S(d) = (r,M) \qquad d \notin \mathsf{dom}(L)}{(\theta,S,R,L) \xrightarrow{(\theta,\mathsf{rd}(d,M))} (\theta,S,R,L[d \mapsto (\mathsf{RO},M)])} \quad \textsc{TxRead2}\ \frac{L(d) = (t,M)}{(\theta,S,R,L) \xrightarrow{(\theta,\mathsf{rd}(d,M))} (\theta,S,R,L)}$$

$$\textsc{TxWrite1}\ \frac{}{(\theta,S,R,L) \xrightarrow{(\theta,\mathsf{wr}(d,M))} (\theta,S,R,L[d \mapsto (\mathsf{Mod},M)])} \quad \textsc{CommitR}\ \frac{L(d) = (\mathsf{RO},M)}{(\mathsf{com}(\theta),S,R,L) \xrightarrow{\epsilon} (\mathsf{com}(\theta),S,R,L_{/d})}$$

$$\textsc{CommitW}\ \frac{S(d) = (r,M) \qquad L(d) = (\mathsf{Mod},M')}{(\mathsf{com}(\theta),S,R,L) \xrightarrow{\epsilon} (\mathsf{com}(\theta),S[d \mapsto (r,M')],R,L_{/d})} \quad \textsc{EndCommit}\ \frac{}{(\mathsf{com}(\theta),S,R,[]) \xrightarrow{\epsilon} (\cdot,S,R,[])}$$

$$\boxed{H;P \xrightarrow{\sigma} H';P'} \quad \textsc{Mutual}\ \frac{H \xrightarrow{(\theta,a)} H' \qquad P \xrightarrow{(\theta,a)} P'}{H;P \xrightarrow{(\theta,a)} H';P'} \quad \textsc{Heap}\ \frac{H \xrightarrow{\epsilon} H'}{H;P \xrightarrow{\epsilon} H';P}$$

Figure 7. Weak semantics (MakeU, MakeRO, and MakeTL rules omitted for space).

3.5 Separation Discipline

The Weak semantics as-is does not provide strong atomicity to all programs. We use the Strong semantics to define the *dynamic separation discipline* [2]. Programs conforming to this discipline are guaranteed to execute with strongly-atomic semantics.

Definition 1 (Separation Discipline)**.** *A program P_0 obeys the* separation discipline *(*$\mathsf{separation}(P_0)$*) if, when* $(\cdot,[],[]);P_0 \rightarrow\!* (s, S,R);P \xrightarrow{(\theta,a)} (s,S',R);P'$ *in the Strong semantics, where* $a = \mathsf{rd}(d,M)$ *or* $\mathsf{wr}(d,M)$*,* $S(d) = (r,M')$ *and* $R(r) = p$*, then:*

- *If* $p = \mathsf{pr}$*, then* $s = \theta$*.*
- *If* $p = \mathsf{unpr}$*, then* $s = \cdot$*.*
- *If* $p = \mathsf{ro}$*, then* $a = \mathsf{rd}(d,M)$*.*
- *If* $p = \mathsf{tl}(\theta')$*, then* $\theta = \theta'$*.*

3.6 Equivalence

We have proved in Coq that the dynamic separation discipline is sufficient for strongly-atomic semantics. First, we establish that the set of program states reachable from programs that obey dynamic separation in the Strong semantics is a superset of the set of program states reachable from the Weak semantics.

Theorem 1. *If* $\mathsf{separation}(P_0)$ *and* $(\cdot,[],[],[]);P_0 \rightarrow\!* (s,S,R,L);$ *P in the Weak semantics, then there exist s', R' and S' such that* $(\cdot,[],[]);P_0 \rightarrow\!* (s',R',S');P$ *in the Strong semantics.*

Theorem 2 will show that the reverse also holds, and therefore that the set of reachable program states is the same for both heap semantics. In order to *prove* both theorems, we must define a correspondence between Weak heaps (s,S,R,L) and Strong heaps

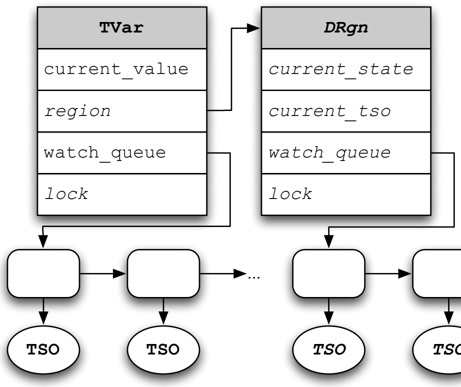

Figure 8. TVar and DRgn structures (new components italicized).

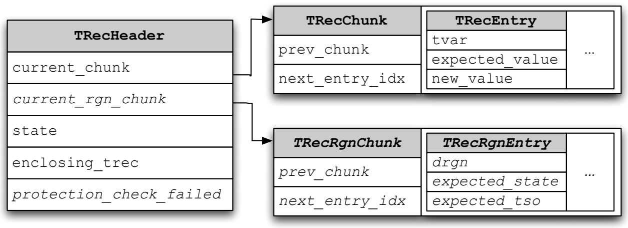

Figure 9. TRec structures (new components italicized).

(s', R', S') (e.g., $R = R'$ at every step). However, the theorem *statement* does not include the correspondence between heaps, since it is sufficient to require that the program states are identical.

The proof of Theorem 1 uses an intermediate semantics in which no other threads can execute while a transaction is active, but transactions still use a lazy-versioning implementation. This semantics, called StrongLazy, is defined by slightly modifying the Weak semantics such that the active and notTx judgments are more strict. The proof is technically interesting, because we must reorder nontransactional pure, rgn, new, read, write, and fork steps such that they do not execute during a transaction's execution or its commit. Such reordering is sound if the program obeys the dynamic separation discipline. We inductively prove soundness by extending the separation discipline to the StrongLazy and Weak semantics.

The other direction is much easier to prove and holds for all programs, not just those that obey dynamic separation:

Theorem 2. *If* $(\cdot, [], []); P_0 \rightarrow* (s, S, R); P$ *in the Strong semantics, then there exist* s', S', R' *and* L' *such that* $(\cdot, [], [], []); P_0 \rightarrow* (s', S', R', L'); P$ *in the Weak semantics.*

4. Implementation

This section describes our modifications to STM Haskell to support dynamic separation. STM Haskell is a lazy-versioning, lazy-conflict-detection STM. In our implementation, nontransactional DVar reads and writes are "weak": they do not interact with the STM. This means the DVar is not locked during these operations. If the program does not obey the dynamic separation discipline defined in Section 3.5, these weak operations may violate the atomicity and isolation of transactions. To preserve strong semantics in programs that obey the discipline, functions like protectDRgn and unprotectDRgn interact with the STM. Specifically, such functions lock their DRgn argument, waiting until any transactions using that DRgn (or any of its DVars) have completed or aborted.

TVars and DVars TVars are C structs with the layout depicted in Figure 8. For convenience, our implementation reuses the same structure for DVars; we will refer to such structs as TVars in this section even though they may be DVars. A TVar holds a pointer to the TVar's current value and a pointer to a *watch queue* of thread state objects (TSOs) that need notification when the TVar's value is updated. We add two fields to the TVar structure (italicized in the figure): a pointer to a DRgn and an explicit lock field. In the original implementation, the current_value pointer doubles as a lock. Because nontransactional TVar operations do not lock the TVar, we implement the lock as an integer in a different field so that nontransactional reads do not observe an ill-typed value.

Regions Regions are represented with the DRgn struct in Figure 8. Like TVars, DRgns hold a lock and a watch queue of threads waiting for the DRgn's state to change.[5] Unlike TVars, DRgns do not hold a pointer to a value; instead, they hold an integer representing the current protection state and an optional pointer to a thread state object (for the thread-local state). It is helpful to think of DRgns as TVars whose "value" is a protection state.

Transaction records While a thread is executing a transaction, it keeps a private record (Figure 9—again, italicized fields have been added by us) of all TVars accessed during the transaction. The record, or TRec, logs for each TVar a record entry containing the TVar's *expected value* (the value the TVar held when first touched by the transaction) and the TVar's *new value* (the current value in the transaction). We have modified transaction records to include a set of accessed DRgns. The record entry for a DRgn records the *expected state* (the protection state the DRgn held when first touched). A boolean field in the TRecHeader indicates whether the transaction has retried due to an inconsistency in protection state; this field is used when deciding whether to restart a transaction.

Transactional operations When a thread performs a readDVar operation, it checks the current TSO to see if it is inside a transaction. If not, it returns the TVar's current value; if so, it incorporates the TVar and the TVar's region into its transaction record, then checks the transaction record to see if the region's protection state allows access. If this check fails, the thread behaves as if it encountered a retry: it adds the TSO to the watch queues of all touched TVars and DRgns, then blocks. writeDVar is similar.

Transaction commit We modified the commit mechanism such that it checks the DRgns touched by the transaction in addition to the TVars. The commit fails (and the transaction retries) if any TVar values or DRgn protection states have changed since the transaction began, or if any TVars or DRgns are locked. Else we update the modified TVars and wake any threads on their watch queues.

Protection state changes Changing a DRgn's protection state is straightforward: acquire the DRgn's writer lock, make the update, wake any threads waiting on that DRgn, and unlock the DRgn. Because transactions acquire a reader lock on all DRgns whose TVars they touch, the protection-state change will block until any concurrently committing transactions have committed or aborted.

5. Evaluation

We have evaluated our implementation on a suite of STM Haskell programs [16]. Figure 10 gives the results of our benchmarks. We ran two versions of each benchmark, one using the original STM Haskell (STM) and one using dynamic separation (DS). The machine was a 4-core 2.8GHz Intel Xeon with 16GB of RAM, running Linux. The speedup for each benchmark is the time for the STM version divided by the time for the DS version.

[5] We actually use reader/writer locks for DRgns so that transactions that access different TVars in the same region may commit without conflicts.

	1 thread			2 threads			4 threads		
Benchmark	STM	DS	Speedup	STM	DS	Speedup	STM	DS	Speedup
TCache	1.30s	1.26s	1.0	1.82s	1.80s	1.0	2.50s	2.50s	1.0
Parallel Sudoku	1.74s	1.75s	1.0	1.07s	1.14s	0.94	0.633s	0.625s	1.0
Blockworld (CCHR)	6.26s	6.93s	0.90	4.23s	4.27s	0.99	7.39s	8.20s	0.90
Prime (CCHR)	28.1s	37.8s	0.74	15.0s	20.1s	0.75	10.6s	13.3s	0.80
Sudoku (CCHR)	0.342s	0.357s	0.96	0.473s	0.528s	0.90	0.591s	0.363s	1.6
Union Find (CCHR)	1.08s	1.27s	0.854	0.808s	0.857s	0.94	0.531s	0.560s	0.95
Shared Int	0.0824s	0.0950s	0.87	0.126s	0.162s	0.78	0.239s	0.517s	0.46
Linked List (build)	0.828s	0.236s	3.5	0.783s	0.210s	3.7	0.766s	0.218s	3.5
Linked List (access)	2.67s	2.67s	1.0	2.06s	2.01s	1.0	1.56s	1.69s	0.92
Linked List (total)	3.50s	2.91s	1.2	2.84s	2.22s	1.3	2.32s	1.91s	1.2
Binary Tree (build)	3.88s	0.453s	8.6	3.84s	0.427s	9.0	3.87s	0.451s	8.6
Binary Tree (access)	18.0s	17.6s	1.0	10.1s	9.91s	1.0	6.14s	5.79s	1.1
Binary Tree (total)	21.9s	18.1s	1.2	13.9s	10.3s	1.3	10.0s	6.24s	1.6
Hash Table	2.64s	1.78s	1.5	2.13s	1.21s	1.8	1.86s	0.929s	2.0

Figure 10. Benchmark results for the original STM Haskell implementation and our implementation of dynamic separation.

The first six lines in Figure 10 are realistic STM Haskell applications. (CCHR is a shared constraint handling library.) For these programs, we changed every `TVar` access to a protected `DVar` access in order to measure the added overhead of regions and protection states. The results show that the dynamic separation version often performs the same as the STM Haskell version, occasionally doing worse but never getting a speedup lower than 0.74.

The bottom half of Figure 10 is a set of microbenchmarks that we examined more closely. Shared Int is a pathological case of high contention: all threads repeatedly try to update a single `TVar`/`DVar`. In this case, we perform significantly worse than STM Haskell, with speedups as low as 0.46 for 4 threads.

The next two benchmarks implement an ordered linked list and a binary search tree using `TVars`. For each benchmark, we changed the `TVars` to `DVars`, and put the entire structure in a single region. We also divided the benchmark into two phases: build and access. In the access phase, the threads perform a total of 25,000 inserts, deletes, and lookups, at rates of 10%, 10%, and 80% respectively. The build phase is a single-threaded initialization, which inserts 2500 elements into the structure. For STM Haskell, we implemented this phase as one large transaction (one transaction per insert had comparable performance). The dynamic separation version does the initialization without using a transaction, then protects the structure's region before spawning threads for the access phase. We achieve speedups between 3.5 and 9.0 for the build phase, confirming that unprotected operations can save significant time over transactions. For the access phase, our implementation is even with STM Haskell, giving a total speedup between 1.2 and 1.6.

The final line in Figure 10 is a new microbenchmark, adapted from an existing hash table implementation [13], that is designed to demonstrate the strengths of dynamic separation. We implemented a hash table that stores key/value pairs in an array of `TVars`/`DVars`. The array dynamically resizes whenever its size hits a threshold, requiring all keys in the table to be rehashed and leading to high contention. To solve this contention problem, we added a conceptual lock, implemented with a `TVar`, that signals whether a rehash is in progress. All hash table operations check the rehash lock before accessing the array, and block if a rehash is in progress. This is more sophisticated than a simple mutex: when no rehash is in progress, inserts, deletes and lookups operate in parallel using STM. In the dynamic separation version, all `DVars` in the table are allocated in the same region. The rehash operation acquires the rehash lock, then unprotects the table's region and does its work without using a transaction. When the rehash completes, it re-protects the table's region and releases the rehash lock. The actual benchmarking code performs 100,000 total operations, divided evenly among inserts, deletes, and lookups. Dynamic separation is very well-suited to this application, achieving speedups of up to 2.0.

6. Related Work

Clearly this work builds most closely on STM Haskell [9, 10] and dynamic separation [2, 3]. Prior work on dynamic separation has been in the context of C# and Automatic Mutual Exclusion [12]. As with our work, the key formal results for dynamic separation and earlier work on static separation [1, 15] involve demonstrating that weakly-atomic and strongly-atomic STMs are indistinguishable for programs obeying the separation discipline. These proofs also use an intermediate semantics to establish the key equivalence result, but all assume an eager-update STM.

While separation puts additional burden on programmers (either to obey a static discipline or insert the right protection-state operations for a dynamic discipline), it avoids semantic problems with weakly-atomic systems. As several prior papers have catalogued [11, 14, 18, 19], naively bypassing STM mechanisms for nontransactional memory accesses leads to semantic anomalies that cannot be explained in terms of data races. Hence the initially imprecise term "weak" atomicity [6] has been refined with more precise notions of what semantic guarantees a TM system provides [7, 14]. Using variants of separation to achieve weaker-but-well-defined semantics would be interesting future work.

Grouping data into regions so that the protection state of the group can change with a single operation is related to an enormous body of work in ownership type systems, region-based memory management, etc. Of particular relevance is recent work on SharC [4, 5], a system that detects concurrency errors in lock-based C programs. Many of the protections states in SharC have analogues in our work, but the implementation details are essentially different due to the speculative (abort-and-retry) nature of STM.

7. Conclusion

Dynamic separation is a flexible and explicit approach to managing transactional and nontransactional access to shared memory. We have designed, implemented, and formalized a dynamic-separation extension to STM Haskell. Our most important innovation has been using regions to change the protection states for multiple objects with one operation. More generally, Haskell has provided an ideal setting to validate and precisely define our work. We hope our work will not only enrich STM Haskell but also influence other languages seeking well-defined meaning for efficient transactions.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[2] M. Abadi, T. Harris, and K. F. Moore. A model of dynamic separation for transactional memory. In *International Conference on Concurrency Theory*, 2008.

[3] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard. Implementation and use of transactional memory with dynamic separation. In *International Conference on Compiler Construction*, 2009.

[4] Z. R. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[5] Z. R. Anderson, D. Gay, and M. Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), Nov. 2006.

[7] L. Dalessandro and M. L. Scott. Strong isolation is a weak idea. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2009.

[8] L. Effinger-Dean and D. Grossman. Modular metatheory for memory consistency models. Technical Report UW-CSE-11-02-01, University of Washington Department of Computer Science & Engineering, 2011.

[9] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *ACM SIGPLAN Haskell Workshop*, 2005.

[10] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[11] R. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *International Symposium on Memory Management*, 2006.

[12] M. Isard and A. Birrell. Automatic mutual exclusion. In *11th Workshop on Hot Topics in Operating Systems*, 2007.

[13] E. Kmett. The Comonad.Reader. `http://comonad.com/reader/`.

[14] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *ACM Symposium on Parallellism in Algorithms and Architectures*, 2008.

[15] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[16] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *ACM International Conference on Computing Frontiers*, 2008.

[17] S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In C. Hoare, M. Broy, and R. Steinbrueggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.

[18] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[19] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report 915, Computer Science Dept., Univ. of Rochester, 2007.