REGION-BASED DYNAMIC SEPARATION FOR STM HASKELL

Laura Effinger-Dean and Dan Grossman
University of Washington
TRANSACT, San Jose, CA, June 5, 2011
Motivation

• We want STM to be correct and fast
• Do nontransactional accesses interact with STM?
  • Yes – reasonable behavior, but slow (strong atomicity)
  • No – fast but has strange behavior (weak atomicity)
• This work focuses on one compromise: dynamic separation
  • Programmer inserts calls to STM when sharing behavior of data changes
Background: Transactional races

- Growing consensus: the following code is racy and therefore the assertion might fail

```c
Thread 1
x = 2;

Thread 2
atomic {
    r1 = x;
    r2 = x;
    assert (r1 == r2);
}
```
Background: Privatization

- Some non-racy idioms are unsafe in basic weak STMs
- Canonical privatization example:

Initially $\text{ptr} \to f = \text{ptr} \to g$

```c
Thread 1
atomic {
  r = \text{ptr};
  \text{ptr} = \text{new} \text{ C}();
}
assert(r \to f == r \to g);
```

```c
Thread 2
atomic {
  \text{++ptr} \to f;
  \text{++ptr} \to g;
}
```

```
\text{ptr}
\text{r}
```

```
\text{f}
\text{g}
```
Background: Privatization

- Eager update: assert sees update from zombie
- Lazy update: assert sees a partially committed transaction

Initially $\text{ptr} \rightarrow f = \text{ptr} \rightarrow g$

```
Thread 1
atomic {
  r = ptr;
  ptr = new C();
}
assert(r->f == r->g);
```

```
Thread 2
atomic {
  ++ptr->f;
  ++ptr->g;
}
```

- Symmetric problem: publication
Existing solutions

- Special-case the privatization and publication idioms
- Support “single global lock atomicity”
- Require programs to obey a separation discipline
  - Separate objects into “always accessed in transactions” and “never accessed in transactions” (and other useful categories)
  - Weak implementation is correct for these programs
  - Static and dynamic approaches
- Our work is on making the dynamic approach more convenient and expressive
**Background: Static separation**

- Type system separates objects into “always accessed in transactions” and “never accessed in transactions”
- State changes like privatization now illegal:

  ```
  Thread 1
  atomic {
    r = ptr;
    ptr = new C();
  }
  assert(r->f == r->g);
  
  Thread 2
  atomic {
    ++ptr->f;
    ++ptr->g;
  }
  ```

Initially \( \text{ptr} \rightarrow f = \text{ptr} \rightarrow g \)

Does not typecheck – \( r \)'s target is accessed both inside and outside transactions

[HMPJH05, ABHI08, MG08]
Background: Dynamic separation

• Every object has a *dynamic protection state*
• Programmer manually instruments state changes

Initially \( \text{ptr} \rightarrow f = \text{ptr} \rightarrow g \)

```
Thread 1
atomic {
    r = ptr;
    ptr = new C();
}
unprotect(r);
assert(r->f == r->g);
```

```
Thread 2
atomic {
    ++ptr->f;
    ++ptr->g;
}
```

Wait for any active transactions to complete

[AHM08, ABHHI09]
Our contributions

• Most important: Dynamic regions allow constant-time state changes for shared data structures
• Extended set of protection states for variables
  • Read-only, thread-local
• Static and dynamic separation exist side-by-side
  • Libraries can be agnostic with respect to transactions
• Formal semantics and proof of correctness
• Haskell implementation & evaluation
So why Haskell?

• Static separation is a natural fit for Haskell
• Add dynamic separation in order to handle cases where static separation is insufficient
  • E.g., privatization
• Existing formal semantics and benchmark suite
STM Haskell

- Static separation
  - Nontransactional variables are IORefs
  - Transactional variables are TVars
- Impossible to use IORefs inside an atomic block
  - Natural application of Haskell’s monadic type system
- Clean semantics for transactions:
  - Sequential and alternative composition
  - Exception handling
  - Manual retry

[HMPJH05]
STM actions

- Actions on TVars can be composed to form STM actions
- STM actions are executed via “atomically”

Example

```
atomically (do {
  t <- newTVar 5;
  x <- readTVar t;
  writeTVar t (x + 1)
})
```

Interface

```
data STM a

data TVar a

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

atomically :: STM a -> IO a
```
Adding dynamic separation

• We will introduce our extensions to STM Haskell’s interface one at a time:
  1. New variable type for dynamic separation: DVars
  2. How to execute dynamic-separation code
  3. New protection states: read-only and thread-local
  4. Shared protection states via regions
1. Adding DVars

- Now three variable types: IORef, TVar, DVar
- DVars correspond to normal variables in languages without static separation

### Static separation interface

```haskell
data STM a
data TVar a

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

### Dynamic separation interface

```haskell
data DSTM a
data DVar a

newDVar :: a -> DSTM (DVar a)
readDVar :: DVar a -> DSTM a
writeDVar :: DVar a -> a -> DSTM a

protectDVar :: DVar a -> IO ()
unprotectDVar :: DVar a -> IO ()
```
2. Running dynamic separation code

- **protected** converts “DSTM action” to “STM action”
  - execute transaction using “atomically (protected (…))”
  - or, seamlessly combined with other STM actions, e.g. using standard sequential composition:

  ```
  Execution example
  do {
    x <- readTVar t;
    protected (writeDVar d x)}
  ```

- **unprotected** converts “DSTM action” to “IO action”
  - runs as a non-transaction
3. Extra protection states

- Prior work included three protection states
  - protected (always-accessed-in-transactions)
  - unprotected (never-accessed-in-transactions)
  - read-only
- We add a fourth state: thread-local

<table>
<thead>
<tr>
<th>Protection state interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>protectDVar</td>
</tr>
<tr>
<td>:: DVar a -&gt; IO ()</td>
</tr>
<tr>
<td>unprotectDVar</td>
</tr>
<tr>
<td>:: DVar a -&gt; IO ()</td>
</tr>
<tr>
<td>makeReadOnlyDVar</td>
</tr>
<tr>
<td>:: DVar a -&gt; IO ()</td>
</tr>
<tr>
<td>makeThreadLocalDVar</td>
</tr>
<tr>
<td>:: DVar a -&gt; IO ()</td>
</tr>
</tbody>
</table>
4. Regions

- Key feature: share protection state across objects in a single data structure
  - Each DVar is allocated in a DRgn
  - Constant-time protection state changes
Final interface

Dynamic separation interface

```haskell
data DSTM a
data DRgn
data DVar a

newDRgn :: DSTM DRgn
newDVar :: a -> DRgn -> DSTM (DVar a)
readDVar :: DVar a -> DSTM a
writeDVar :: DVar a -> a -> DSTM a

protectDRgn :: DRgn -> IO ()
unprotectDRgn :: DRgn -> IO ()
makeReadOnlyDRgn :: DRgn -> IO ()
makeThreadLocalDRgn :: DRgn -> IO ()

protected :: DSTM a -> STM a
unprotected :: DSTM a -> IO a
```
Example: hash table

- Many DVars, allocated in a single DRgn
- Usually “protected”: multiple threads concurrently perform inserts, lookups, and deletes using transactions
- Rehashing of all keys causes high contention: do in isolation (no concurrent operations)
- Implement rehash as non-transaction for efficiency, “unprotecting” the table’s DRgn during rehash
Example: hash table

Dynamic separation example

```
insert :: DHash k v -> k -> v -> IO ()
insert hash key value = do {
  doRehash <- atomically (...);
  when doRehash (do {
    unprotectDRgn (rgn hash);
    unprotected (do {
      table <- readDVar (dvar hash);
      table' <- DHT.rehash table;
      writeDVar (dvar hash) table' });
    protectDRgn (rgn hash);
    atomically (...) });
  atomically (insertHelper hash key value) ;
```

Check table size and acquire table lock.
Unprotect table’s DRgn.
Perform rehash as non-transaction.
Re-protect table’s DRgn.
Release lock.
Do insert.
Formalism

- We define two sets of formal semantics:
  1. **Strong**: describes a strongly-atomic STM.
  2. **Weak**: describes a lazy-update weakly-atomic STM.

- Novel to our work:
  - Formalism extended to include regions, thread-local, read-only
  - Weak semantics uses lazy- rather than eager-update
  - All proofs verified in Coq

- See paper for more details
Theoretical Results

“Easy” theorem:

• Strong => Weak

(Much, much) harder theorem:

• Weak => Strong (for well-behaved programs only)
Well-behaved?

- A program is well-behaved if, under the Strong semantics, a thread executes a read or write of a DVar and the DVar’s DRgn’s protection state is:
  1. protected – the thread must be in a transaction
  2. unprotected – the thread must NOT be in a transaction
  3. read-only – the action must be a read
  4. thread-local to $\theta$ – the thread’s ID must be $\theta$
Implementation

- Protection state change is mini-transaction
- readDVar and writeDVar check the DRgn’s protection state and abort if inconsistent
- See paper for more details
Performance results

• Approach 1: Take existing STM benchmarks, use all DVars instead of TVars, measure slowdown: 0-20%
  • Caveat: STM Haskell is not state-of-the-art
• Approach 2: Code up “killer uses” of dynamic separation, measure speedup: 2-8x for 4 threads
  • e.g., initializing a data structure, resizing a hash table
• Approach 3: Find an STM Haskell program that would benefit from dynamic separation and rewrite it: future work
Conclusion

• Dynamic separation is an elegant way of addressing the semantic issues with weakly-atomic STM implementations
• We enrich existing work on dynamic separation by adding regions and useful new protection states
• Novel Haskell interface allows static- and dynamic-separation code to be used side-by-side
• We also provide a rigorous formalism and prototype implementation