Region-Based Dynamic Separation in STM Haskell
(And Related Perspective)

Dan Grossman

University of Washington

Transactional Memory Workshop

April 30, 2010

Apology

AIR	**Thursday, 29APR 2010**		
	Alaska Airlines	**Flight Number**: 22	Class: K-Coach/Economy
	From: Seattle/Tacoma WA, USA	**Depart**: 12:25 PM	
	To: Chicago O'Hare IL, USA	**Arrive**: 06:24 PM	
	Stops: 0	Duration: 3 hour(s) 59 minute(s)	
	Seats: 26D	Status: CONFIRMED	Miles: 1721
	Equipment: Boeing 737 Jet	MEAL: FOOD TO PURCHASE	
	ARRIVES ORD TERMINAL 3 Frequent Flyer Number: AS94211950 -		
	Alaska Airlines Confirmation number is IJFPYS Check in on-line for Alaska		
CAR	**Thursday, 29APR 2010**		

From: Hank Levy (Department Chair)

Date: April 6, 2010

Subject: Upcoming faculty meetings

… **Please reserve ** NOON TO 5:30 PM ** on THURSDAY APRIL 29th for a possible (marathon) faculty meeting…**

Apology

AIR	Thursday, 29APR 2010		
	United Airlines	**Flight Number**: 754	Class: W-Coach/Economy
	From: Seattle/Tacoma WA, USA	**Depart**: 11:15 PM	
	To: Chicago O'Hare IL, USA	**Arrive**: 05:03 AM 30APR	
	Stops: 0	Duration: 3 hour(s) 48 minute(s)	
	Seats: 23C	Status: CONFIRMED	Miles: 1721
	Equipment: Airbus A320 Jet	MEAL: MEAL AT COST	
	ARRIVES ORD TERMINAL 1		
	United Airlines Confirmation number is XLZCC8 Check in on-line for United		
CAR	Friday, 30APR 2010		

From: Nicholas Kidd

Subject: Re: [TMW'10] A few announcements

Ugh indeed, this sounds terrible …

I hereby promise that coffee will be available throughout TMW'10!

TM at Univ. Washington

I come at transactions from the programming-languages side

- Formal semantics, language design, and efficient implementation for atomic blocks
- Software-development benefits
- Interaction with other sophisticated features of modern PLs

[ICFP05][MSPC06][PLDI07][OOPSLA07][SCHEME07][POPL08]

```
transfer(from,to,amt){
  atomic {
    deposit(to,amt);
    withdraw(from,amt);
  }
}
```

An *easier-to-use* and *harder-to-implement* synchronization primitive

The goal

I want atomic blocks to:

- Be easy to use in most cases
- Interact well with rest of language design / implementation
  - Despite subtle semantic issues for PL experts

My favorite analogy [OOPSLA07] : garbage collection is a success story, for memory management rather than concurrency

- People forget subtle semantic issues exist for GC
  - Finalization / resurrection
  - Space-explosion "optimizations" (like removing **`x=null`**)
  - …

Today

- Review and perspective on transaction + non-transaction access
  - "How we got to where we are"
  - A healthy reminder, probably without (much) controversy
  - But not much new for this expert crowd
- Not-yet-published work on specific issue of *dynamic separation*
  - Extension of STM Haskell
  - Emphasize need for "regions" and libraries reusable inside and outside transactions
- Time permitting: Brief note on two other current projects

Are races allowed?

For performance and legacy reasons, many experts have decided *not* to allow code like the following

```
Thread 1

x = 2;
```

```
Thread 2
atomic {
  x = 1;
  y = 1;
  assert(x==y);
}
```

- I can probably grudgingly live with this
  - Why penalize “good code” for questionable benefit
- But:
  - For managed PLs, still struggle with “what can happen”
  - Does make it harder to maintain / evolve code

Privatization

Alas, there are examples where it is awkward to consider the program racy, but “basic” TM approaches can “create” a problem

Canonical “privatization” example:

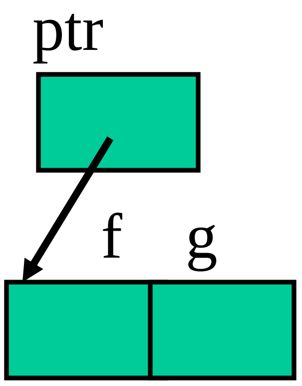
```
initially ptr.f == ptr.g
```

Thread 1

```
atomic {
  r = ptr;
  ptr = new C();
}
assert(r.f==r.g);
```

Thread 2

```
atomic {
  ++ptr.f;
  ++ptr.g;
}
```



The Problems

Eager update, lazy conflict detection:

`assert` may see one update from "doomed" Thread 2

Lazy update:

`assert` may see one update from "partially committed" Thread 2

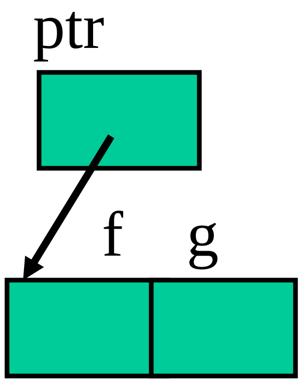
```
initially ptr.f == ptr.g
```

```
Thread 1

atomic {
  r = ptr;
  ptr = new C();
}
assert(r.f==r.g);
```

```
Thread 2

atomic {
  ++ptr.f;
  ++ptr.g;
}
```



Solution areas

To support atomic blocks that privatize (and related idioms):

1. Enrich underlying TM implementations to be privatization safe
   - I’m all for it if trade-offs are acceptable
     - Important but uncommon cases
   - Not today’s presentation

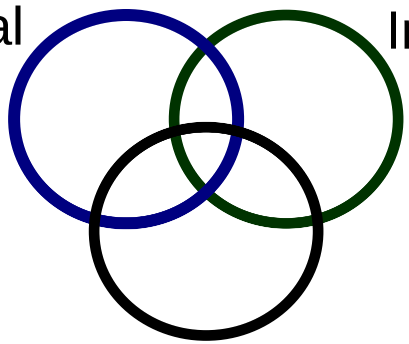
2. Disallow privatization
   - Either soundly prohibited by PL or programmer error

3. Allow privatization only if programmers do more explicit work
   - Our work, making this more convenient and flexible

Disallowing privatization

Prior work on static separation takes this approach

- Same memory cannot be used inside a transaction and outside a transaction
- Note read-only and thread-local are okay



See:

- NAIT is provably enough for "weak" TM to implement "strong" atomic block
  - POPL08 * 2
- STM Haskell
  - functional + monads
    => immutable or NAIT

Dynamic separation

Dynamic separation allows objects to transition among

- Only accessed inside transactions
- Only accessed outside transactions
- Read only
- (Added by us: thread-local to thread **`tid`**)

Explicit language primitives to enact transitions

- Example: **`protect obj`** transitions **`obj`** to "only inside"

Semantics and implementation for C# and AME

- [Abadi et al, CC2009, CONCUR2008]

Uses of dynamic separation

- Obvious use: Explicit privatization

- Another: more efficient (re)-initialization of data structures than static separation would allow
  - Essentially a "publication"
  - Create a large tree in one thread without transactions and then **`protect`** it and make it thread-shared
  - Resize a hashtable without a long transaction (next slide)

- But the (re)-initialization argument is much more compelling if we can transition an entire data structure in $O(1)$ time/space
  - For example: If hash table uses linked lists

Hash table example

```
class HT {
  T [] table;
  boolean resizing = false;
  …
  void insert(T x){ atomic{ if(resizing) retry; … }}
  T find(int key) { atomic{ if(resizing) retry; … }}
  void resize() {
     atomic{ if(resizing) return; resizing = true; }
     unprotect(table);
     …
     protect(table);
     atomic{ resizing = false; }
  }
}
```

Today

Laura
Effinger-Dean

- Review and perspective on transaction + non-transaction access
  - “How we got to where we are”
  - A healthy reminder, probably without (much) controversy
  - But not much new for this expert crowd

- Not-yet-published work on specific issue of *dynamic separation*
  - Extension of STM Haskell
  - Emphasize need for “regions” and libraries reusable inside and outside transactions

- Time permitting: Brief note on two other current projects

Why Haskell

- In some sense, Haskell is a terrible choice for dynamic separation
  - The one language where static separation is natural
  - Monads already enforce static separation of many things

- But this makes it an ideal setting for our research
  - Use dynamic separation only where static separation is unpalatable
  - Need a precise, workable semantics from the start, else it will be obvious we are "ruining Haskell"

Novelties

1. Region-based to support constant-time transition-change for collection of objects

2. Complement static separation (current default in Haskell)
   – Allow both approaches in same program (different data)
   – Use dynamic separation for composable libraries that can be used inside or outside transactions, without violating Haskell's type system

3. Extend elegant formal semantics (including **`orelse`**)

4. Underlying implementation uses lazy update
   – Significant speed-up for some benchmarks by avoiding transactions that are necessary with static separation

STM Haskell basics

STM Haskell has static separation

- Most data is read-only (purely functional language)
- Non-transactional mutable locations called **`IORef`**s
- Transactional mutable locations called **`TVar`**s

Because the type system enforces static separation, you can't "transactionalize" code using **`IORef`**s, by "slapping an atomic around it"

- This is a general feature of Haskell's monads
- The *STM monad* and *IO (top-level) monad* are distinct
- atomically primitive takes a transaction "object" and creates a top-level-action "object"

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

Adding DVars

From a language-design standpoint, it's mostly straightforward to add a third kind of mutable location for dynamic separation

- In "normal languages", a **`DVar`** would be allowed by the type system to be accessed anywhere
  - A meta-data field would record "current protection state" and dynamically disallow transactions to use it when "unprotected"
  - This doesn't work with monads: separation is the rule

DVars for Haskell

- So we add a third monad, *DSTM monad*, for **`Dvar`**s
  - Can turns a DSTM "object" into an STM "object" or a top-level-action "object"

```
atomically  :: STM a  -> IO a
protected   :: DSTM a -> STM a
unprotected :: DSTM a -> IO a   -- not atomic!
```

- A DSTM "object" could be as little as a single read/write of a **`DVar`**
  - But sequences of actions can be packaged up so that the same library can be used inside or outside transactions
  - Trade-off between code reuse and protection-state checks
  - Not possible in previous approaches to sound separation

Regions

So far, we could just have the DSTM Monad include operations, including protection-state changes for **`DVar`**s

```
newDRgn        :: DSTM DRgn
                  a -> DRgn -> DSTM (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 ()
protectDRgn    :: DRgn -> IO ()
unprotectDRgn  :: DRgn -> IO ()
```

Instead, we add a level of indirection for the protection state, so one state change can effect a collection of objects (could be 1)

– Cost is one implicit word per **`DVar`** (avoidable if unneeded)

Novelties

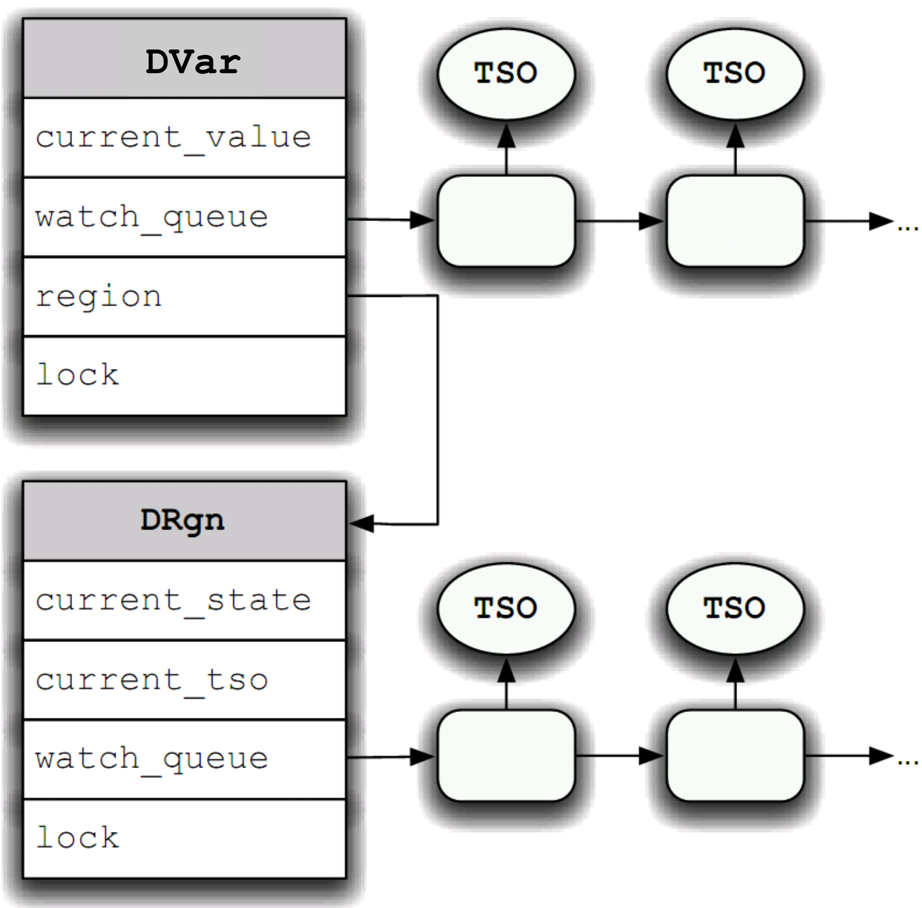
1. Region-based to support constant-time transition-change for collection of objects

2. Complement static separation (current default in Haskell)
   – Allow both approaches in same program (different data)
   – Use dynamic separation for composable libraries that can be used inside or outside transactions, without violating Haskell's type system

3. Extend elegant formal semantics (including `orelse`)

4. Underlying implementation uses lazy update
   – Significant speed-up for some benchmarks by avoiding transactions that are necessary with static separation

Implementation in one slide



- **`DVar`** read/write also reads associated **`DRgn`**
  - Only txn's first access of the **`DVar`** (easy with lazy update)
- Protection-state change is a mini-transaction that writes to the **`DRgn`**
  - TM mechanism synchronizes with txns
- There are, uhm, some other details ☺

Non-transactional accesses

- Suppose **`DVar`** accesses outside of transactions do not check the **`DRgn`** protection-state
  - Any correct program w.r.t. dynamic separation runs correctly
  - Any incorrect program is still type safe, but may violate atomicity

- Alternately, we can check all accesses
  - Have a safe caching mechanism to avoid unnecessary **`DRgn`** access in common cases

Preliminary Performance

Caveat: Comparing to STM Haskell baseline is not necessarily state-of-the-art

- Approach 1: Take existing STM benchmarks, use all **`DVar`**s instead of **`TVar`**s, measure slowdown: 0-20%
- Approach 2: Code up “killer uses” of dynamic separation, measure speedup: 2-8x for 4 threads (e.g., resizing hash table)
- Approach 3: Find an STM Haskell program that would benefit from dynamic separation and rewrite it: TBD

Conclusion

Dynamic separation appears to be an elegant and viable alternative for implementing a PL over a TM that is not privatization-safe