Reagents: expressing and composing fine-grained concurrency

Aaron Turon
Northeastern University
CAS: cost versus contention

Throughput
Sequential

Conention (log-scale)

Threads

0.2%
0.25%
0.33%
0.5%
1%
2%
100%

2
4
6
8
Synchronization
Reentrant locks
Semaphores
R/W locks
Reentrant R/W locks
Condition variables
Countdown latches
Cyclic barriers
Phasers
Exchangers

Data structures
Queues
  Nonblocking
  Blocking (array & list)
Synchronous
Priority, nonblocking
Priority, blocking
Deques
Sets
Maps (hash & skiplist)
class TreiberStack[A] {
    private val head =
        new AtomicRef[List[A]](Nil)

    def push(a: A) {
        val backoff = new Backoff
        while (true) {
            val cur = head.get()
            if (head.cas(cur, a :: cur)) return backoff.once()
        }
    }
}
Head

3 → 2
Head

5 → 3 → 2

7
CAS fail
Head

[Diagram of a linked list with nodes labeled 7, 5, 3, 2]
def tryPop():  Option[A] = {
    val backoff = new Backoff
    while (true) {
        val cur = head.get()
        cur match {
            case Nil     => return None
            case a::tail =>
                if (head.cas(cur, tail))
                    return Some(a)
        }
        backoff.once()
    }
}
Concurrency libraries are indispensable, but hard to build and extend.
The Proposal:
Scalable concurrent algorithms can be built and extended using abstraction and composition.
Design
Lambda: the ultimate abstraction
Lambda: the ultimate abstraction
Lambda: the ultimate abstraction
Lambda abstraction:
Lambda abstraction:

Reagent abstraction:
\( c : \text{ Chan}[A,B] \)
c: Chan[A, B]
c: Chan[A, B]
Message passing

\[ \text{swap} \]
Message passing

\[
\text{swap} \quad (A, B) \rightarrow (A, C)
\]

\[
r: \text{Ref}[A] \\
f: (A, B) \rightarrow (A, C)
\]
Message passing

Shared state

swap

upd

f

A

B

R

A

B

S

A

B
Message passing

shared state

A B
Message passing

Shared state

Disjunction
Message passing

Shared state

Disjunction
Message passing

- Swap

Shared state

- Upd

Disjunction

- $R + S$

A

- $R * S$

$(B, C)$
Message passing

- swap

Shared state

- upd
  - f

Disjunction

- R
  - +
  - S

Conjunction

- R
  - *
  - S
2-way join

\[(A, B)\]

\[
\begin{array}{c}
\text{swap} \\
A \rightarrow C
\end{array}
\quad \times \quad
\begin{array}{c}
\text{swap} \\
B \rightarrow d
\end{array}
\]
2-way join

(A, B)

\((A, B)\)

\(*\)

\(\text{swap}\)

\(A \rightarrow c\)

\(B \rightarrow d\)

\(\text{Exn} \rightarrow e\)

\(\text{swap}\)
Abortable 2-way join

\[(A, B)\]

\[
\begin{array}{c}
\text{swap} \\
A \downarrow c \\
\end{array} 
\quad \ast \quad 
\begin{array}{c}
\text{swap} \\
B \downarrow d \\
\end{array}
\quad + 
\begin{array}{c}
\text{swap} \\
\text{Exn} \downarrow e \\
\end{array}
\]
class TreiberStack [A] {
    private val head = new Ref[List[A]](Nil)
    val push = upd(head)(cons)
    val tryPop = upd(head) {
        case (x :: xs) => (xs, Some(x))
        case Nil => (Nil, None)
    }
}
```scala
class TreiberStack [A] {
    private val head = new Ref[List[A]](Nil)
    val push = upd(head)(cons)
    val tryPop = upd(head) {
        case (x :: xs) => (xs, Some(x))
        case Nil       => (Nil, None)
    }
    val pop = upd(head) {
        case (x :: xs) => (xs, x)
    }
}
```
class TreiberStack [A] {
    private val head = new Ref[List[A]](Nil)
    val push = upd(head)(cons)
    val tryPop = upd(head)(trySplit)
    val pop = upd(head)(split)
}
```scala
class TreiberStack [A] {
  private val head = new Ref[List[A]](Nil)
  val push = upd(head)(cons)
  val tryPop = upd(head)(trySplit)
  val pop = upd(head)(split)
}

class EliminationStack [A] {
  private val stack = new TreiberStack[A]
  private val (send, recv) = new Chan[A]
  val push = stack.push + swap(send)
  val pop = stack.pop + swap(recv)
}
```
stack1.pop >> stack2.push
Implementation
Phase 1
Accumulate CASes
Phase 2
Phase 1
Accumulate CASes

Phase 2
Attempt k-CAS
Accumulate CASes  Attempt $k$-CAS
Permanent failure

Accumulate CASes  Attempt k-CAS
Permanent failure

Accumulate CASes

Attempt k-CAS

Transient failure
Permanent failure
Permanent failure

? failure

Transient failure

Transient failure
Permanent failure

Transient failure

? failure

Permanent failure

Transient failure

\[ P \& P = P \]
\[ T \& T = T \]
\[ P \& T = T \]
\[ T \& P = T \]
Is this just STM?
Is this just STM?

No:

- Single CAS collapses to single phase
- Multiple CASes to single location forbidden
  
  So the “redo log” is write-only for phase 1

Therefore: pay-as-you-go

- Treiber stack is really a Treiber stack
- Pay for kCAS only for compositions
Is this just STM?

- Isolation
  - Shared state

- Interaction
  - Message passing
Is this just STM?

**Isolation**
- Shared state

**Interaction**
- Message passing

Using lock-free bags, based on earlier work with Russo [OOPSLA’11]
6.1 Concurrent ML

Originally, CML was focused on managing concurrency rather than profiting from parallelism, and this focus was reflected in structures while providing scalable composed operations that are simpler than the stack or queue we give in our elimination stack or Michael-Scott queue (some evidence for higher-order concurrency).

Reagents carve out a middle ground between completely hand-built concurrent data sharing and the completely automatic written algorithms and the completely automatic.

The results are shown in Figure 6. The x-axes show thread counts.

Benchmarking results

Reagents can plausibly compete with hand-built concurrent data structures, while providing scalable composed operations that are easier to write and easier to reason about than the stack or queue.

6.2 Software transactional memory

Parallel CML uses lock-based queues to store messages, but where Parallel CML uses locks to store messages, Reagents can plausibly compete with hand-built concurrent data structures.

Figure 6. The results are shown in Figure 6. The x-axes show thread counts.

The results are shown in the figure. The x-axis represents the number of threads, and the y-axis represents throughput in iterations per microsecond. The figure shows the performance of different approaches: Reagent-based, Hand-build, Lock-based, and STM-based. The Reagent-based approach shows the best performance, followed by Hand-build, Lock-based, and STM-based.

Throughput

An event making a choice is enrolled as offering a communication of additional combinators, including those dealing directly with events, and include variants of CML's core event combinators. But abstractly as functions, it is impossible to express the choice of two abstractly as functions, it is impossible to express the choice of two.
Throughput - iterations/s is bigger is better.

It is addressed by fine-grained concurrent data structures and synchronization protocols. Reagents are designed for writing and tailoring events and include variants of CML's core event combinators. But the lock or STM-based data structures. The results show that the lock or STM-based data structures.

CML was proposed. The key challenge is resolving uses of its implementation. More recently, a parallel implementation of where CML is aimed squarely at capturing synchronous communication. Reagents are clearly influenced by the design of CML's abstract protocols. The solution is...
java.util.concurrent

**Synchronization**
- Reentrant locks
- Semaphores
- R/W locks
- Reentrant R/W locks
- Condition variables
- Countdown latches
- Cyclic barriers
- Phasers
- Exchangers

**Data structures**
- Queues
  - Nonblocking
  - Blocking (array & list)
- Synchronous
- Priority, nonblocking
- Priority, blocking
- Deques
- Sets
- Maps (hash & skiplist)
Synchronization
- Reentrant locks
- Semaphores
- R/W locks
- Reentrant R/W locks
- Condition variables
- Countdown latches
- Cyclic barriers
- Phasers
- Exchangers

Data structures
- Queues
  - Nonblocking
  - Blocking
  - Synchronous
- Priority, nonblocking
- Priority, blocking
- Deques
- Sets
- Maps (hash & skiplist)
The take-away:
Reagents enable scalable concurrent algorithms to be built and extended using abstraction and composition

https://github.com/aturon/ChemistrySet