Reagents: Expressing and Composing Fine-grained Concurrency

Aaron Turon
Northeastern University
turon@ccs.neu.edu

Abstract

Efficient communication and synchronization is crucial for fine-grained parallelism. Libraries providing such features, while indispensable, are difficult to write, and often cannot be tailored or composed to meet the needs of specific users. We introduce *reagents*, a set of combinators for concisely expressing concurrency algorithms. Reagents scale as well as their hand-coded counterparts, while providing the composability existing libraries lack.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming; D.3.3 [*Language constructs and features*]: Concurrent programming structures

General Terms Design, Algorithms, Languages, Performance

Keywords fine-grained concurrency, nonblocking algorithms, monads, arrows, compositional concurrency

1. Introduction

Programs are what happens between cache misses.

The problem

Amdahl's law tells us that sequential bottlenecks fundamentally limit our profit from parallelism. In practice, the effect is amplified by another factor: interprocessor communication, often in the form of cache coherence. When one thread waits on another, the program pays the cost of lost parallelism *and* an extra cache miss. The extra misses can easily accumulate to yield parallel *slowdown*, more than negating the benefits of the remaining parallelism.

Cache, as ever, is king.

The easy answer is: avoid communication. In other words, parallelize at a coarse grain, giving threads large chunks of independent work. But some work doesn't easily factor into large chunks, or equal-size chunks. Fine-grained parallelism is easier to find, and easier to sprinkle throughout existing sequential code.

Another answer is: communicate efficiently. The past two decades have produced a sizable collection of algorithms for synchronization, communication, and shared storage which minimize the use of memory bandwidth and avoid unnecessary waiting. This research effort has led to industrial-strength libraries—`java.util.concurrent` (JUC) the most prominent—offering a wide range of primitives appropriate for fine-grained parallelism.

Such libraries are an enormous undertaking—and one that must be repeated for new platforms. They tend to be conservative, implementing only those data structures and primitives likely to fulfill common needs, and it is generally not possible to safely combine the facilities of the library. For example, JUC provides queues, sets and maps, but not stacks or bags. Its queues come in both blocking and nonblocking forms, while its sets and maps are nonblocking only. Although the queues provide atomic (thread-safe) dequeuing and sets provide atomic insertion, it is not possible to combine these into a single atomic operation that moves an element from a queue into a set.

In short, libraries for fine-grained concurrency are indispensable, but hard to write, hard to extend by composition, and hard to tailor to the needs of particular users.

Our contribution

We have developed *reagents*, which abstractly represent fine-grained concurrent operations. Reagents are *expressive* and *composable*, but when invoked retain the scalability and performance of existing algorithms:

Expressive Reagents provide a basic set of building blocks for writing concurrent data structures and synchronizers. The building blocks include isolated atomic updates to shared state, and interactive synchronous communication through message passing. This blend of isolation and interaction is essential for expressing the full range of fine-grained concurrent algorithms (§3.3). The building blocks also bake in many common concurrency patterns, like optimistic retry loops, backoff schemes, and blocking and signaling. Using reagents, it is possible to express sophisticated concurrency algorithms at a higher level of abstraction, while retaining the performance and scalability of direct, hand-written implementations.

Composable Reagents also provide a set of composition operators, including choice, sequencing and pairing. Using these operators, clients can extend and tailor concurrency primitives without knowledge of the underlying algorithms. For example, if a reagent provides only a nonblocking version of an operation like `dequeue`, a user can easily tailor it to a version that blocks if the queue is empty; this extension will work regardless of how `dequeue` is defined, and will continue to work even if the `dequeue` implementation is changed. Similarly, clients of reagent libraries can sequence operations together to provide atomic move operations between arbitrary concurrent collections—again, without access to or knowledge of the implementations. Compositionality is also useful to algorithm designers, since many sophisticated algorithms can be understood as compositions of simpler ones (§3.3, §3.4).

We begin in §2 by illustrating, through examples, some of the common patterns, tradeoffs and concerns in designing fine-grained concurrent algorithms. The two subsequent sections contain our core technical contributions:

- The **design** of the reagent combinators is given in §3. Each combinator is motivated by specific needs and patterns in concurrent algorithms; the section shows in particular how to write all of the algorithms described in §2 concisely and at a higher-than-usual level of abstraction.
- The **implementation** of reagents is detailed in §5, via both high-level discussion and excerpts from our code in Scala. It reveals the extent to which reagents turn patterns of fine-grained concurrency into a general algorithmic framework. It also shows how certain choices in the design of reagents enable important optimizations in their implementation.

Reagents have a clear cost model that provides an important guarantee: they do not impose extra overhead on the atomic updates performed by individual concurrent algorithms. That is, a reagent-based algorithm manipulates shared memory in exactly the same way as its hand-written counterpart, even if it reads and writes many disparate locations. This is a guarantee not generally provided by concurrency abstractions like software transactional memory (STM), which employ redo or undo logs or other mechanisms to support composable atomicity. Reagents only impose overheads when *multiple* algorithms are sequenced into a large atomic block, *e.g.* in an atomic transfer between collections.

In principle, then, reagents offer a strictly better situation than with current libraries: when used to express the algorithms provided by current libraries, reagents provide a higher level of abstraction yet impose negligible overhead; nothing is lost. But unlike with current libraries, the algorithms can then be extended, tailored, and combined. Extra costs are only paid when the new atomic compositions are used.

We test this principle empirically in §6 by comparing multiple reagent-based collections to their hand-written counterparts, as well as to lock-based and STM-based implementations. The benchmarks include both single operations used in isolation (where little overhead for reagents is expected) and combined atomic transfers. We compare at both high and low levels of contention. Reagents perform universally better than the lock- and STM-based implementations, and are indeed competitive with hand-written lock-free implementations.

Finally, it is worth noting in passing that reagents generalize both Concurrent ML [22] and Transactional Events [4], yielding the first lock-free implementation for both. We give a thorough discussion of this and other related work—especially STM—in §7.

A prototype implementation of reagents, together with the benchmarks and benchmarking harness we used, can be found at

`https://github.com/aturon/ChemistrySet`

2. Background

Broadly, we are interested in data structures and algorithms for communication, synchronization, or both. This section gives a brief survey of the most important techniques for communication and synchronization in a fine-grained setting—exactly the techniques that reagents abstract and generalize. Readers already familiar with Treiber stacks [28], elimination-backoff stacks [9], dual stacks [24], and MCS locks [19] can safely skip to §3.

Given our target of cache-coherent, shared-memory architectures, the most direct way of communicating between threads is modifying shared memory. The challenge is to provide both *atomicity* and *scalability*: communications must happen concurrently both without corruption and without clogging the limited memory bandwidth, even when many cores are communicating. A simple way to provide atomicity is to associate a lock with each shared data structure, acquiring the lock before performing any operation.

```
class TreiberStack[A] {
  private val head = new AtomicReference[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()
    }
  }
  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()
    }
  }
}
```

Figure 1. Treiber's stack (in Scala)

The prevailing wisdom[1] is that such *coarse-grained* locking is inherently unscalable:

- It forces operations on the data structure to be serialized, even when they could be performed in parallel.
- It adds extra cache-coherence traffic, since each core must acquire the same lock's cache line in exclusive mode before operating. For fine-grained communication, that means at least one cache miss per operation.
- It is susceptible to preemptions or stalls of a thread holding a lock, which prevents other threads from making progress.

To achieve scalability, data structures employ finer-grained locking, or eschew locking altogether. Fine-grained locking associates locks with small, independent parts of a data structure, allowing those parts to be manipulated in parallel. Lockless (or *nonblocking*) data structures instead perform updates directly, using hardware-level operations (like compare-and-set) to ensure atomicity. Doing the updates directly means that there is no extra communication or contention for locks, but it also generally means that the entire update must consist of changing a single word of memory—a constraint which is often quite challenging to meet.

Fig. 1 gives a classic example of a fine-grained concurrent data structure: Treiber's lock-free stack [28]. The stack is represented as an *immutable* linked list. Mutation occurs solely through the head pointer, represented here as an AtomicReference. The head is mutated using compareAndSet (here abbreviated as cas), which takes an expected value and a new value, and atomically updates the reference if it has the expected value. It returns **true** iff the update was successful.

A word about Scala: the syntax

exp **match** { **case** pat1 ⇒ body1 ... **case** patN ⇒ bodyN }

denotes pattern matching, where the value of exp is matched in order against pat1 up to patN. In Fig. 1, we use the two list constructors Nil and :: (an inline cons) in patterns.

[1] With some dissenting opinions, as in the recent work on *flat combining* [10].

The push and tryPop operations are implemented in a typical *optimistic* style: they take a snapshot of the head, perform some local computation with it, and then attempt to update it accordingly. The computation is optimistic because it is performed without holding a lock. The head might concurrently change, invalidating the cas intended to update it. To account for possible interference, the snapshot-compute-update code executes within a retry loop. While the loop may retry forever, it can only do so by repeatedly failing to cas, which in turn means that an unlimited number of other operations on the stack are succeeding. In other words, push and tryPop are formally lock free.

To successfully cas, a core must acquire the relevant cache line in exclusive mode, or something conceptually equivalent. When several cores attempt to cas a common reference, they must coordinate at the hardware level, using precious memory bandwidth and wasting processor cycles. A failed cas is evidence that there is *contention* over a common location. To increase the cas success rate and conserve memory bandwidth, fine-grained concurrent algorithms employ a *backoff* scheme, which here we have abstracted into a lightweight Backoff class. The class encapsulates the simplest scheme: busy-wait for a random amount of time that grows exponentially each time once is invoked.

Treiber's stack scales better than a lock-based stack mainly because it decreases the amount of shared data that must be updated per operation: instead of acquiring a shared lock, altering a shared stack pointer, and releasing the shared lock, Treiber's stack does a single CAS to update the shared pointer directly. However, that pointer is still a centralized source of contention. While exponential backoff helps relieve the contention, we can do better by parallelizing stack operations—which is counterintuitive, given that they all involve modifying the top of the stack.

Parallelization requires a change of perspective. Normally, we view concurrent operations on the same part of a data structure as competing to atomically update that data structure; we want the operations to be isolated. On the other hand, sometimes operations can "help" each other: a push and a tryPop effectively cancel each other out. This insight leads to a scheme called *elimination backoff* [9, 26]. Operations first try the usual cas-based code. If the cas fails, rather than busy-waiting, the operations advertise their presence on a side-channel, reducing the contention for the head pointer. If a push and tryPop detect their mutual presence, the push can pass its argument directly to tryPop through the side-channel, and no change to the head pointer is necessary. Atomicity is not violated, because had the push and pop executed in sequence, the head pointer would have been returned to its original value anyway. On the other hand, if no dual operation is detected during backoff, the operation withdraws its offer and tries once more to cas the head.

Both push and tryPop are *total* operations: they can succeed no matter what state the stack is in, and fail (and retry) only due to active interference from concurrent threads. A true pop operation, on the other hand, is *partial*: it is undefined when the stack is empty. Often this is taken to mean that the operation should block until another thread moves the stack into a state on which the operation is defined. Partial operations introduce considerable complexity, because *all* operations on the data structure must potentially signal blocked threads, depending on the changes being performed. In some cases, it is possible to cleverly treat signaling in essentially the same way as atomic updates [24].

Synchronization and signaling is also subject to cache and memory bandwidth concerns, but it would take us too far afield to discuss these in depth. Mellor-Crummey and Scott pioneered the now-common techniques for grappling with these concerns [19], and we apply their techniques in implementing the blocking and signaling protocols for reagents (§5).

```
// Shared state
upd:   Ref[A] ⇒ (A × B ⇀ A × C) ⇒ Reagent[B,C]

// Message passing
swap: Endpoint[A,B] ⇒ Reagent[A,B]

// Composition
+ :     Reagent[A,B] × Reagent[A,B] ⇒ Reagent[A,B]
>> :    Reagent[A,B] × Reagent[B,C] ⇒ Reagent[A,C]
* :     Reagent[A,B] × Reagent[A,C] ⇒ Reagent[A, B × C]

// Post−commit actions
postCommit: (A ⇒ Unit) ⇒ Reagent[A,A]
```

Figure 2. The core reagent combinators

3. Reagents: the core combinators

Reagents are a new instance of an old idea: representing computations as data. The computations being represented are fine-grained concurrent operations, so a value of type Reagent[A,B] represents a function from A to B that internally interacts with a concurrent data structure through mutation, synchronization, or both. Because the computations are data, however, they can be combined in ways that go beyond simple function composition. Each way of combining reagents corresponds to a way of combining their internal interactions with concurrent data structures. Existing reagents—for example, those built by a concurrency expert—can be composed by library users, without those users knowing their internal implementation. This way of balancing *abstraction* and *composition* was pioneered with Concurrent ML [22], and is now associated with *monads* [30] and *arrows* [14]. Our contribution is giving a set of combinators appropriate for expressing and composing fine-grained concurrent algorithms, with a clear cost semantics and implementation story (§5).

The reagent combinators encompass a blend of duals:

Shared state *versus* Message passing

Given the goal of using reagents to express algorithms like Treiber's stack (§2), it is not surprising that reagents include primitives from shared-state concurrency. But what about message passing? Underlying the shared state/message passing duality is a deeper one:

Isolation *versus* Interaction

Operations on shared state are generally required to be atomic, which means in particular that they are isolated from one another; concurrent shared-state operations appear to be *linearizable* [13] into a series of nonoverlapping, sequential operations. Synchronous message passing is just the opposite: rather than appearing to not overlap, sends and receives are *required* to overlap. Fine-grained concurrent operations straddle these two extremes, appearing to be isolated while internally tolerating (or exploiting) interaction. Elimination backoff (§2) provides a striking example of this phenomenon, and reagents can capture it concisely (§3.3).

There is also duality in reagent composition:

Disjunction *versus* Conjunction

In particular, we can combine two reagents by requiring exactly one of them to take effect (choice), or by requiring both of them to take effect in a single atomic step (join).

Finally, reagents, like functions, are inert. To be useful, they must be invoked. Reagents offer two means of invocation:

Active (reactants) *versus* Passive (catalysts)

In chemistry, a reagent is a participant in a reaction, and reagents are subdivided into *reactants*, which are consumed during reaction, and *catalysts*, which enable reactions but are not consumed by them. Similarly for us. Invoking a reagent as a reactant is akin to calling it as a function: its internal operations are performed once, yielding a result or blocking until it is possible to do so. Invoking it as a catalyst instead makes it passively available as a participant in reactions. Because catalysts are not "used up," they can participate in many reactions in parallel.

In the rest of this section, we introduce the core reagent combinators and use them to build a series of increasingly complex concurrent algorithms. By the end, we will have seen how to implement all of the algorithms described in §2, and several more besides.

3.1 Atomic updates on Refs

Memory is shared between reagents using the type Ref[A] of atomically-updatable references. The upd combinator (Fig. 2) represents atomic updates to references. It takes an *update function*, which tells how to transform a snapshot of the reference cell and some input into an updated value for the cell and some output. Using upd, we can rewrite TreiberStack in a more readable and concise way:

```
class TreiberStack[A] {
  private val head = new Ref[List[A]](Nil)
  val push: Reagent[A, Unit] = upd(head) {
    case (xs, x) ⇒ (x::xs, ())
  }
  val tryPop: Reagent[Unit, Option[A]] = upd(head) {
    case (x::xs, ()) ⇒ (xs, Some(x))
    case (Nil,   ()) ⇒ (Nil, None)
  }
}
```

In Scala, anonymous partial functions are written as a series of cases enclosed in braces. For push and tryPop, the case analyses are exhaustive, so the update functions are in fact total. Unit is akin to void: it is a type with a single member, written ().

Being reagents, push and tryPop are inert values. They can be invoked as reactants using the ! method, which is pronounced "react." For a Reagent[A,B] the ! method takes an A and returns a B. Scala permits infix notation for methods, so we can use a TreiberStack s by writing s.push ! 42. The key point is that when we invoke these reagents, we are executing *exactly* the same algorithms written by hand in §2, including the retry loop with exponential backoff; reagents systematize and internalize common patterns of fine-grained concurrency. By exposing push and tryPop as reagents rather than methods, we allow further composition and tailoring by a user of the data structure (§3.3, §3.4).

While tryPop handles both empty and nonempty stacks, we can write a variant that drops the empty case:

```
val pop: Reagent[Unit, A] =
  upd(head) { case (x::xs, ()) ⇒ (xs, x) }
```

Now our update function is partial. An invocation s.pop ! () will block the calling thread unless or until the stack is nonempty.

In general, there are two ways a reagent can fail to react: *transiently* or *permanently*. Transient failures arise when a reagent loses a race to CAS a location; they can only be caused by *active interference* from another thread. A reagent that has failed transiently should retry, rather than block, following the concurrency patterns laid out in §2. Permanent failures arise when a reagent places requirements on its environment—such as the requirement, with pop above, that the head reference yield a nonempty list. Such failures are permanent in the sense that only activity by another thread can enable the reagent to proceed. When faced with a permanent failure, a reagent should block until signaled that the underlying state has changed. Blocking and signaling are entirely handled by the reagent implementation; there is therefore no risk of lost wakeups.

The reagent upd(f) reagent can fail permanently only for those inputs on which f is undefined.

3.2 Synchronization: how reagents react

With reagents, updates to shared memory are isolated, so they cannot be used for interaction in which the parties are mutually aware. Reagents interact instead through *synchronous swap channels*, which consist of two complementary *endpoints*. The method mkChan[A,B] returns a pair of type

(Endpoint[A,B], Endpoint[B,A])

The combinator for communication is swap (see Fig. 2), which lifts an Endpoint[A,B] to a Reagent[A,B]. When two reagents communicate on opposite endpoints, they provide messages of complementary type (A and B, for example) and receive each other's messages. We call a successful communication a *reaction* between reagents. On the other hand, if no complementary message is available, swap will block until a reaction can take place—a permanent failure.

3.3 Disjunction of reagents: choice

If r and s are two reagents of the same type, their *choice* r + s will behave like one of them, nondeterministically, when invoked. The most straightforward use of choice is waiting on several signals simultaneously, while consuming only one of them. For example, if c and d are endpoints of the same type, swap(c) + swap(d) is a reagent that will accept exactly one message, either from c or from d. If neither endpoint has a message available, the reagent will block until one of them does.

A more interesting use of choice is adding backoff strategies (of the kind described in §2). For example, we can build an elimination backoff stack as follows:

```
class EliminationStack[A] {
  private val s = new TreiberStack[A]
  private val (elimPop, elimPush) = mkChan[Unit,A]
  val push: Reagent[A,Unit] = s.push + swap(elimPush)
  val pop:  Reagent[Unit,A] = s.pop  + swap(elimPop)
}
```

Choice is left-biased, so when push is invoked, it will first attempt to push onto the underlying Treiber stack. If the underlying push fails (due to a lost CAS race), push will attempt to send a message along elimPush, *i.e.*, to synchronize with a concurrent popper. If it succeeds, the push reagent completes without ever having modified its underlying stack.

For choice, failure depends on the underlying reagents. A choice fails permanently only when *both* of its underlying reagents have failed permanently. If either fails transiently, the choice reagent has failed transiently and should therefore retry. Reasoning along these lines, we deduce that push never blocks, since the underlying s.push can only fail transiently. On the other hand, pop can block because s.pop can fail permanently on an empty stack and swap(elimPop) can fail permanently if there are no offers from pushers.

When push or pop retry, they will *spinwait* briefly for another thread to accept their message along elimPush or elimPop; the length of the wait grows exponentially, as part of the exponential backoff logic. Once the waiting time is up, the communication attempt is canceled, and the whole reagent is retried. This protocol is elaborated in §5.

3.4 Conjunction of reagents: sequencing and pairing

Choice offers a kind of disjunction on reagents. There are also two ways of *conjoining* two reagents, so that the composed reagent has the effect of both underlying reagents:

- End-to-end composition, via *sequencing*: if r: Reagent[A,B] and s: Reagent[B,C] then r >> s: Reagent[A,C].
- Side-by-side composition, via *pairing*: if r: Reagent[A,B] and s: Reagent[A,C] then r * s: Reagent[A,(B,C)].

These combinators differ only in information flow. Each guarantees that the atomic actions of both underlying reagents become a single atomic action for the composition. For example, if s1 and s2 are both stacks, then s1.pop >> s2.push is a reagent that will atomically transfer an element from the top of one to the top of the other. The reagent will block if s1 is empty. Similarly, s1.pop * s2.pop will pop, in one atomic action, the top elements of both stacks, or block if either is empty.

Here we begin to see the benefits of the reagent abstraction. Both of the example combinations work regardless of how the underlying stacks are implemented. If both stacks use elimination backoff, the conjoined operations will potentially use elimination on both simultaneously. This behavior is entirely emergent; it does not require any code on the part of the stack author, and it does not require the stack user to know anything about the implementation. Reagents can be composed in unanticipated ways.

Conjunctions provide a solution to the Dining Philosophers problem: to consume two resources atomically, one simply conjoins two reagents that each consume a single resource. For example, if c and d are endpoints of type Unit to A and B respectively, then swap(c) * swap(d) is a reagent that receives messages on both endpoints simultaneously and atomically. There is no risk of introducing a deadlock through inconsistent acquisition ordering, because the reagents implementation is responsible for the ultimately acquisition order, and will ensure that this order is globally consistent.

The failure behavior of conjunctions is dual to that of disjunctions: if *either* conjunct fails permanently, the entire conjunction fails permanently.

The implementation details for conjunctions are discussed later (§5), but a key point is that the performance cost is *pay as you go*. Single atomic reagents like push and pop execute a single CAS—just like the standard nonblocking algorithms they are meant to implement—even though these operations can be combined into larger atomic operations. The cost of conjunction is only incurred when a conjoined reagent is actually used. This is a crucial difference from STM, which generally incurs overheads regardless of the size of the atomic blocks; see §7 for more discussion.

3.5 Catalysts: persistent reagents

The ! operator invokes a reagent as a reactant: the invocation lasts for a single reaction, and any messages sent by the reagent are consumed by the reaction. But sometimes it is useful for reagent invocations to persist beyond a single reaction, *i.e.*, to act as catalysts. For example, the following function creates a catalyst that merges input from two endpoints and sends the resulting pairs to another endpoint:

```
def zip(in1: Endpoint[Unit, A], in2: Endpoint[Unit, B],
        out: Endpoint[(A,B), Unit]) =
  dissolve ((swap(in1) * swap(in2)) >> swap(out))
```

The dissolve function takes a Reagent[Unit, Unit] and introduces it as a catalyst.[2] Operationally, in this example, that just means sending messages along in1 and in2 that are marked as catalyzing messages, and hence are not consumed during reaction. The upshot is that senders along in1 will see the catalyzing messages, look for messages along in2 to pair with, and ultimately send messages along out (and similarly in the other order).

Catalysts could instead be expressed using a thread that repeatedly invokes a reagent as a reactant. Allowing direct expression through dissolve is more efficient (since it does not tie up a thread) and allows greater parallelism (since, as with the zip example above, multiple reagents can react with it in parallel).

Catalysts are not limited to message passing. The zip example above could be rephrased in terms of arbitrary reagents rather than just endpoints.

[2] For simplicity, we have not given a way to cancel catalysts after they have been introduced, but cancellation is easy to add.

3.6 Post-commit actions

Reagents support "post commit actions", which comprise code to be run after a reaction has successfully taken place, *e.g.* for signaling or spawning another thread after an operation completes. The postCommit combinator (Fig. 2) takes a function from A to Unit and produces a Reagent[A,A]. The post-commit action will be recorded along with the input of type A, which is passed along unchanged. Once the reaction completes, the action will be invoked on the stored input. The combinator is meant to be used in sequence with other combinators that will produce the appropriate input. For example, the reagent pop >> postCommit(println) will print the popped element from a stack after the pop has completed.

3.7 Case study: the join calculus

Fournet and Gonthier's *join calculus* [6] provides an interesting example of the expressive power of reagents. The join calculus is based on message passing, but instead of a simple "receive" primitive on channels, programmers write *join patterns*. The join patterns for a set of channels say, once and for all, how to react to messages along those channels. Each pattern $c_1(x_1) \& \cdots \& c_n(x_n) \triangleright b$ consists of a sequence of channels c_i with names x_i for the messages along those channels, and a body b in which the names x_i are bound. A join pattern *matches* if messages are available on each of the listed channels, and it *fires* by atomically consuming *all* of the messages and then running the body. Since several patterns may be given for the same channels, the choice of which pattern to fire may be nondeterministic.

The join pattern $c_1(x_1) \ \& \ \cdots \ \& \ c_n(x_n) \triangleright b$ can be interpreted directly as a catalyst. The join operator & is interpreted as a conjunction *, the channel names as appropriate swap instances, and the body b as a post-commit action. Altogether, the reagent corresponding to the pattern is

$$(\mathsf{swap}(c_1) \ * \ \cdots \ * \ \mathsf{swap}(c_n)) \ \texttt{>>} \ \mathsf{postCommit}(b)$$

A set of join patterns governing a set of channels can all be written in this way and dissolved as catalysts, which is equivalent to dissolving the choice of all the patterns. Thus, reagents provide a scalable implementation of the join calculus, along the lines of the one developed in previous work with Russo [29].

3.8 Atomicity guarantees

Because conjunction distributes over disjunction, every reagent built using the core combinators (Fig. 2) can be viewed as a disjunction of conjunctions, where each conjunction contains some combination of updates and swaps. For such a reagent, reactions atomically execute all of the conjuncts within exactly one of the disjuncts. This STM-like guarantee is too strong for algorithms which read shared memory without requiring the reads to be "visible" (*i.e.*, to participate in an atomic transaction). The next section will introduce *computed* reagents which allow invisible reads and writes, trading weaker guarantees for better performance.

```
// Low−level shared state combinators
read:  Ref[A] ⇒ Reagent[Unit, A]
cas:   Ref[A] × A × A ⇒ Reagent[Unit, Unit]

// Computational combinators
ret:        A ⇒ Reagent[Unit,A]
computed: (A ⇀ Reagent[Unit, B]) ⇒ Reagent[A,B]
```

Figure 3. The low-level and computational combinators

When reagents interact through message passing, their atomicity becomes intertwined: they must react together in a single atomic step. This requirement raises an important but subtle question: what should happen when isolation and interaction conflict? Consider two reagents that interact over a channel, but also each update the *same* shared reference. The atomicity semantics demands that both reagents involved in the reaction atomically commit, but the isolation on references demands that the updates be performed in separate atomic steps.

For both simplicity and performance, we consider such situations to be *illegal*, and throw an exception in such cases. Ideally, the static types of reagents would somehow track the necessary information to determine whether compositions are safe, but we leave such a type discipline for future work. In practice, this rules out only compositions of certain operations within the *same* data structure, which are much less common than compositions across data structures. It is also straightforward to adopt an alternative approach, *e.g.* the one taken by Communicating Transactions (§7), which treats isolation/interaction conflicts as transient failures.

4. Low-level and computational combinators

4.1 Computed reagents

The combinators introduced in §3 are powerful, but they impose a strict phase separation: reagents are constructed prior to, and independently from, the data that flows through them. Phase separation is useful, because it allows reagent execution to be optimized based on complete knowledge of the computation to be performed (see §5). But in many cases the choice of computation to be performed depends on the input or other dynamic data. The computed combinator (Fig. 3) expresses such cases. It takes a partial function from A to Reagent[Unit,B] and yields a Reagent[A,B]. When the reagent computed(f) is invoked, it is given an argument value of type A, to which it applies the function f. If f is not defined for that input, the computed reagent issues a permanent (blocking) failure, similarly to the upd function. Otherwise, the application of f will yield another, dynamically-computed reagent, which is then invoked with (), the unit value.

In functional programming terms, the core reagent combinators of §3 can be viewed in terms of *arrows* [14], which are abstract, composable computations whose structure is statically determined. With the addition of computed, reagents can also be viewed in terms of *monads* [30], which extend arrows with dynamic determination of computational structure.

In the remainder of this section, we introduce a handful of lower-level combinators which are useful in connection with computed reagents. We close with a case study: Michael and Scott's lock-free queue [20].

4.2 Shared state: read and cas

Although the upd combinator is convenient, it is sometimes necessary to work with shared state with a greater degree of control. To this end, we include two combinators, read and cas (see Fig. 2), for working directly on Ref values. Together with the computed combinator described in §4.1, read and cas suffice to *build* upd.

The read combinator is straightforward: if r has type Ref[A], then read(r) has type Reagent[Unit, A] and, when invoked, returns a snapshot of r. The cas combinator takes a Ref[A] and two A arguments, giving the expected and updated values, respectively. Unlike its counterpart for AtomicReference, a cas reagent does *not* yield a boolean result. A failure to CAS is transient, and therefore results in a retry.

```
class MSQueue[A] {
  private case class Node(data: A, next: Ref[Node])
  private val head = new Ref(new Node(null)) // sentinel
  private val tail  = new Ref(read(head) ! ()) // sentinel
  val tryDeq: Reagent[Unit, Option[A]] = upd(head) {
    case (Node(_, Ref(n@Node(x, _))), ()) ⇒ (n, Some(x))
    case (emp,                        ()) ⇒ (emp, None)
  }
  private def findAndEnq(n: Node): Reagent[Unit,Unit] =
    read( tail ) ! () match {
      case ov@Node(_, r@Ref(null)) ⇒ // found true tail
        cas(r, null, n) >>
        postCommit { cas(tail, ov, n)? ! () }
      case ov@Node(_, Ref(nv)) ⇒ // not the true tail
        cas( tail , ov, nv)? ! ();  // catch up tail ref
        findAndEnq(n)
    }
  val enq: Reagent[A, Unit] = computed {
    (x: A) ⇒ findAndEnq(new Node(x, new Ref(null)))
  }
}
```

Figure 4. The Michael-Scott queue, using reagents

4.3 Constant and tentative reagents

The ret combinator (Fig. 2) always succeeds, immediately returning the given value.

Because choice is left-biased, it can be used together with the remaining combinators to express *tentative* reagents: if r is a Reagent[A,B] then r? is a Reagent[A,Option[B]] that first tries r (wrapping its output with Some) and then tries ret(None), which always succeeds. This allows a reaction to be attempted, without retrying it when it fails.

4.4 Case study: the Michael-Scott queue

To illustrate the use of computed, we now show how to implement the classic Michael-Scott lock-free queue [20]. Unlike a stack, in which all activity focuses on the head, queues have two loci of updates. That means, in particular, that the Refs used by its reagents may vary depending on the current state of the queue. The strategy we employ to implement it readily scales to more complicated examples, such as concurrent skiplists or the lazy, lock-free set algorithm [11]. With any of these examples, we reap the usual benefits: a concise, composable and extensible exposition of the algorithm.

Here is a brief overview of the Michael-Scott algorithm. The queue is represented as a *mutable* linked list, with a sentinel node at the head (front) of the queue. The head pointer always points to the current sentinel node; nodes are dequeued by a CAS to this pointer, just like Treiber stacks (but lagged by one node). The true tail of the queue is the unique node, reachable from the head pointer, with a null next pointer; thanks to the sentinel, such a node is guaranteed to exist. If the queue is empty, the same node will be the head (sentinel) and tail. Finally, as an optimization for enqueing, a tail pointer is maintained with the invariant that the true tail node is always reachable from it. The tail pointer may lag behind the true tail node, however, which allows the algorithm to work using only single-word CAS instructions.

Our reagent-based implementation of the Michael-Scott queue is shown in Fig. 4. The node representation is given as an inner *case* class. In Scala, case classes provide two features we take advantage of. First, the parameters to their constructors (here data and next) are automatically added as final fields to the class, which are initialized to the constructor argument values. Second, they extend pattern matching through **case** so that instances can be *deconstructed*. A pattern like **case** Node(d, n) matches any instance of the node class, binding d to its data field and n to its next field.

The head and tail references of the queue are initialized to the same sentinel node. Here we use the read combinator (§4.2) to extract the sentinel value from the head when constructing the tail. The read reagent in tail 's initializing expression is immediately executed during construction of an MSQueue instance.

The tryDeq reagent is very similar to the tryPop reagent in TreiberStack, modulo the sentinel node. The reagent pattern matches on the sentinel node, ignoring its data field by using _, the wildcard. The next field is then matched to a nested pattern, Ref(n@Node(x, _)). This pattern immediately reads the current value of the reference stored in next, binds that value to n, and then matches the pattern Node(x,_) against n. If the pattern matches—which it will any time the next field of the sentinel is non-null—the node n becomes the new head (and hence the new sentinel).

Since the location of the tail node is determined dynamically by the data in the queue, the enq reagent must itself be determined dynamically. For enq, we compute a dynamic reagent by first taking the given input x, creating a node with that data, and then calling a private function findAndEnq that will locate the tail of the queue and yield a reagent to update it to the new node. Since findAndEnq is private and tail-recursive, Scala will compile it to a loop.

The findAndEnq function searches for the *true* tail node (whose next field is null) starting from the tail pointer, which may lag. To perform the search, findAndEnq must read the tail pointer, which it does using the read combinator. *There is a subtle but important point here: this read occurs while the final reagent is being computed.* That means, in particular, that the read is *not* part of the computed reagent; it is a side-effect of computing the reagent. The distinction is important: such a read is effectively "invisible" to the outer reagent being computed, and thus is not guaranteed to happen atomically with it. Invisible reads and writes are useful for avoiding compound atomic updates, but must be employed carefully to ensure that the computed reagent provides appropriate atomicity guarantees.

Once the tail pointer has been read, its value is pattern-matched to determine whether it points to the true tail. If it does, findAndEnq yields a cas reagent (§4.2) that will update the next field of the tail node from null to the new node. The attached post-commit action attempts to catch up the tail pointer through a cas. Since the cas fails only if further nodes have been enqueued by other concurrent threads, we perform it tentatively (§4.3); it is not necessary or desirable to retry on failure.

If, on the other hand, the tail pointer is lagging, findAndEnq performs an invisible cas to update it. Since it may be racing with other enqueuers to catch up the tail, a failure to CAS is ignored here. Regardless of the outcome of the cas, the findAndEnq function will restart from a freshly-read tail pointer. Notice that in this case, an entire iteration of findAndEnq is executed with no visible impact or record on the final computed reagent—there is no extended redo log or compound atomic transaction.

5. Implementation

Having seen the design and application of reagents, we now turn to their implementation. We begin with a high-level overview (§5.1) introducing the key techniques and data structures we use. We then delve into the core code of our Scala implementation, beginning with the primary entry point (the ! method, §5.4) and continuing with the key combinators. For space reasons, we do not discuss the implementation of catalysts, which is fairly straightforward.

5.1 The basic approach

When invoked, reagents attempt to *react*, which is *conceptually* a two phase process: first, the desired reaction is built up; second, the reaction is atomically committed. We emphasize "conceptually" because, as discussed in the introduction, reagents are designed to avoid this kind of overhead in the common case. We first discuss the general case (which imposes overhead) but return momentarily to the common (no overhead) case.

An attempt to react can fail during either phase. A failure during the first phase, *i.e.* a failure to build up the desired reaction, is always a permanent failure (§3.3). Permanent failures indicate that the reagent cannot proceed given current conditions, and should therefore block until another thread intervenes and causes conditions to change. On the other hand, a failure during the second phase, *i.e.* a failure to commit, is always a transient failure (§3.3). Transient failures indicate that the reagent should retry, since the reaction was halted due to active interference from another thread. In general, a reaction encompasses three lists: the CASes to be performed, the messages to be consumed, and the actions to be performed after committing. It thus resembles the redo log used in some STM implementations [15].

In the common case that a reagent performs only one visible (§4.1) CAS or message swap, those components of the reaction are not necessary and hence are not used. Instead, the CAS or swap is performed immediately, compressing the two phases of reaction. Aside from avoiding extra allocations, this key optimization means that in the common case a cas or upd in a reagent leads to exactly one executed CAS during reaction, with no extra overhead. When a reaction encompasses multiple visible CASes, a costlier *k*CAS protocol must be used to ensure atomicity. We discuss the *k*CAS protocol in §5.5, and the common case single CAS in §5.6.

In the implementation, Reagent[A,B] is an abstract class all of whose subclasses are hidden. The subclasses roughly correspond to the combinator functions (which are responsible for instantiating them), and instances of the subclasses store the arguments given to their corresponding combinator. Each subclass provides an implementation of the tryReact method, which is an abstract method of Reagent[A,B] with the following signature:

```
def tryReact(a: A, rx: Reaction, offer: Offer[B]): Any
```

The Any type in Scala lies at the top of the subtyping hierarchy, akin to Object in Java. Here we are using Any to represent a union of the type B with the type Failure, where the latter has just two singleton instances, Block and Retry, corresponding to permanent and transient failures.

In other words, tryReact takes the input (type A) to the reagent and the reaction built up so far, and either completes the reaction, returning a result (type B), or fails, returning one of the failure singletons (Block or Retry). The remaining argument, offer, is used for synchronization and communication between reagents, which we explain next.

5.2 Offers

Message passing between reagents is synchronous, meaning that both reagents take part in a single, common reaction. In the implementation, this works by one reagent placing an *offer* to react

```
def !(a: A): B = {
  val backoff = new Backoff
  def withoutOffer(): B =
    tryReact(a, empty, null) match {
      case Block ⇒ withOffer()
      case Retry ⇒
        backoff.once()
        if (maySync) withOffer() else withoutOffer()
      case ans   ⇒ ans.asInstanceOf[B]
    }
  def withOffer(): B = {
    val offer = new Offer[B]
    tryReact(a, empty, offer) match {
      case (f: Failure) ⇒
        if (f == Block) park() else backoff.once(offer)
        if (offer.rescind) withOffer() else offer.answer
      case ans ⇒ ans.asInstanceOf[B]
    }
  }
  withoutOffer()
}
```

Figure 5. The ! method, defined in Reagent[A,B]

in a location visible to the other. The reagent making the offer either spinwaits or blocks until the offer is fulfilled; if it spinwaits, it may later decide to withdraw the offer. The reagent accepting the offer combines the accumulated reactions of both reagents, and attempts to commit them together. Fulfilling the offer means, in particular, providing a final "answer" value that should be returned by the reagent that made the offer. Each offer includes a status field, which is either Pending, Rescinded, or a final answer. Hence, the Offer class is parameterized by the answer type; a Reagent[A,B] will use Offer[B]. When fulfilling an offer, a reagent CASes its status from Pending to the desired final answer.

In addition to providing a basic means of synchronization, the offer data structure is used to resolve external choices. For example, the reagent swap(ep1) + swap(ep2) may resolve its choices internally by fulfilling an existing offer on ep1 or ep2; but if no offers are available, the reagent will post a *single* offer to *both* endpoints, allowing the choice to be resolved externally. Reagents attempting to consume that offer will race to change a single, shared status field, thereby ensuring that such choices are resolved atomically.

Offers are made as part of the same tryReact process that builds and commits reactions. The offer argument to tryReact is non-null whenever an offer is to be made.

5.3 Continuations

For implementing backtracking choice and message passing, it is necessary for each reagent to know and have control over the reagents that are sequenced after it. Thus we do not represent the reagent sequencing combinator >> with its own class. Instead, each reagent records its own *continuation*, which is another reagent. Thus, for example, while the cas combinator produces a reagent of type Reagent[Unit,Unit], the CAS class has a parameter k of type Reagent[Unit,R], and CAS extends Reagent[Unit,R] rather than Reagent[Unit,Unit]. The R stands for (final) result.

The combinator functions are responsible for mapping from the user-facing API, which does not use continuations, to the internal reagent subclasses, which do. Each reagent initially begins with the empty continuation, called Commit, the behavior of which is explained in §5.5. The sequencing combinator then merely plumbs together the continuation arguments of its parameters.

5.4 The entry point: reacting

The code for performing a reaction is given in the ! method definition for Reagent[A,B], shown in Fig. 5. This method provides two generalized versions of the optimistic retry loops we described in §2. The retry loops are written as a local, tail-recursive functions, which Scala compiles down to loops.

The first retry loop, withoutOffer, attempts to perform the reaction without making visible offers to other reagents. It may, however, find and consume offers from other reagents as necessary for message passing. To initiate the reaction, withoutOffer calls the abstract tryReact method with the input a, an empty reaction to start with, and no offer. If the reaction fails in the first phase (a permanent failure, represented by Block), the next attempt must be made with an offer, to set up the blocking/signaling protocol. If the reaction fails in the second phase (a transient failure, represented by Retry), there is likely contention over shared data. To reduce the contention, withoutOffer performs one cycle of exponential backoff before retrying. If the reagent includes communication attempts, the retry is performed with an offer, since doing so increases chances of elimination (§3.3) without further contention. Finally, if both phases of the reaction succeed, the final answer is returned.

The second retry loop, withOffer, is similar, but begins by allocating an Offer object to make visible to other reagents. Once the offer had been made, the reagent can actually block when faced with a permanent failure; the offer will ensure that the attempted reaction is visible to other reagents, which may complete it. Blocking is performed by the park method provided by Java's LockSupport class. On a transient failure, the reagent spinwaits, checking the offer's status. In either case, once the reagent has finished waiting it attempts to rescind the offer, which will fail if another reagent has fulfilled the offer.[3]

Initially, the reaction is attempted using withoutOffer, representing optimism that the reaction can be completed without making a visible offer.

5.5 The exit point: committing

As mentioned in §5.2, the initial continuation for reagents is the Commit continuation, shown in Fig. 6. The tryReact method of Commit makes the transition from building up a Reaction object to actually committing it.

If the reagent has made an offer, but has also completed the first phase of reaction, the offer must be rescinded before the commit phase is attempted—otherwise, the reaction could complete twice. As with the ! method, the attempt to rescind the offer is in a race with other reagents that may be completing the offer. If Commit loses the race, it returns the answer provided by the offer. Otherwise, it attempts to commit the reaction, and if successful simply returns its input, which is the final answer for the reaction.

Committing a reaction requires a kCAS operation: k compare and sets must be performed atomically. This operation, which forms the basis of STM, is in general expensive and not available through hardware. There are several software implementations that provide nonblocking progress guarantees [1, 7, 18]. Reagents that perform a multiword CAS will inherit the progress properties of the chosen implementation.

For our prototype implementation, we have opted to use an extremely simple implementation that replaces each location to be CASed with a sentinel value, essentially locking the location. As the Reaction object is assembled, locations are kept in address order, which guarantees a consistent global order and hence avoids dead- and live-lock within the kCAS implementation. The advan-

[3] Even if the reagent had blocked, it is still necessary to check the status of its offer, because park allows spurious wakeups.

```
class Commit[A] extends Reagent[A,A] {
  def tryReact(a: A, rx: Reaction, offer: Offer[A]) =
    if (offer != null && !offer.rescind) offer.answer
    else if (rx.commit) a
    else Retry
}
```

```
class CAS[A,R](ref: Ref[A], ov: A, nv: A, k: Reagent[A,R])
  extends Reagent[Unit,R] {
  def tryReact(u: Unit, rx: Reaction, offer: Offer[R]) =
    if (!rx.hasCAS && !k.hasCAS)
      if (ref.cas(ov, nv)) k.tryReact((), rx, offer)
      else Retry
    else
      k.tryReact((), rx.withCAS(ref, ov, nv), offer)
}
```

```
class Choice[A,B](r1: Reagent[A,B], r2: Reagent[A,B])
  extends Reagent[A,B] {
  def tryReact(a: A, rx: Reaction, offer: Offer[B]) =
    r1.tryReact(a, rx, offer) match {
      case Retry ⇒ r2.tryReact(a, rx, offer) match {
        case (_: Failure) ⇒ Retry // must retry r1
        case ans          ⇒ ans
      }
      case Block ⇒ r2.tryReact(a, rx, offer)
      case ans   ⇒ ans
    }
}
```

```
class Computed[A,B](c: A ⇀ Reagent[Unit,B])
  extends Reagent[A,B] {
  def tryReact(a: A, rx: Reaction, offer: Offer[B]) =
    if (c.isDefinedAt(a)) c(a).tryReact((), rx, offer)
    else Block
}
```

```
class Swap[A,B,R](ep: Endpoint[A,B], k: Reagent[B, R])
  extends Reagent[A,R] {
  // NB: this code glosses over some important details
  //      discussed in the Channels subsection
  def tryReact(a: A, rx: Reaction, offer: Offer[R]) = {
    if (offer != null) // send message if so requested
      ep.put(new Message(a, rx, k, offer))
    def tryFrom(cur: Cursor, failMode: Failure): Any =
      cur.getNext match {
        case Some(msg, next) ⇒
          msg.exchange(k).tryReact(a, rx, offer) match {
            case Retry ⇒ tryFrom(next, Retry)
            case Block ⇒ tryFrom(next, failMode)
            case ans   ⇒ ans
          }
        case None ⇒ failMode
      }
    tryFrom(ep.dual.cursor, Retry) // attempt reaction
  }
```

Figure 6. Excerpts from the reagent subclasses

tage of this implementation, other than its simplicity, is that is has no impact on the performance of single-word CASes to references, which we expect to be the common case; such CASes can be performed directly, without any awareness of the kCAS protocol. Our experimental results in §6 indicate that even this simple kCAS implementation provides reasonable performance—much better than STM or coarse-grained locking—but a more sophisticated kCAS would likely do even better.

5.6 Shared state

The implementation of the cas combinator is given by the CAS class, shown in Fig. 6. Its tryReact method is fairly simple, but it illustrates a key optimization we have mentioned several times: if neither the reaction so far nor the continuation of the cas are performing a CAS, then the entire reagent is performing a single CAS, and can thus attempt the CAS immediately. This optimization eliminates the overhead of creating a new Reaction object and employing the kCAS protocol, and it means that lock-free algorithms like TreiberStack and MSQueue behave just like their hand-written counterparts. If, on the other hand, the reagent may perform a kCAS, then the current cas is recorded into a new Reaction object, which is passed to the continuation. In either case, the continuation is invoked with the unit value as its argument.

5.7 Choice

The implementation of choice (Fig. 6) is pleasantly simple. It attempts a reaction with either arm of the choice, going in left to right order. As explained in §3.3, a permanent failure of choice can only result from a permanent failure of *both* arms. Also, note that the right arm is tried even if the left arm has only failed transiently.

5.8 Computed reagents

The implementation of computed reagents (Fig. 6) is exactly as described in §4.1: attempt to execute the stored computation c on the argument a to the reagent, and invoke the resulting reagent with a unit value. If c is not defined at a, the computed reagent issues a permanent failure. The implementation makes clear that the invisible reads and writes performed within the computation c do not even have access to the Reaction object, and so cannot enlarge the atomic update performed when it is committed.

5.9 Channels

We represent each endpoint of a channel as a lock-free bag (which can itself be built using reagents). The lock-freedom allows multiple reagents to interact with the bag in parallel; the fact that it is a bag rather than a queue trades a weaker ordering guarantee for increased parallelism, but any lock-free collection would suffice.

The endpoint bags store messages, which wrap offers with additional data from the sender:

```
case class Message[A,B,C](
  payload: A,             // sender's actual message data
  senderRx: Reaction,     // sender's checkpointed reaction
  senderK: Reagent[B,C],  // sender's reagent continuation
  offer: Offer[C])        // sender's offer
```

Each message is essentially a checkpoint of a reaction in progress, where the reaction is blocked until the payload (of type A) can be swapped for a dual payload (of type B). Hence the stored sender continuation takes a B for input; it returns a C, which matches the final answer type of the sender's offer.

The core implementation of swap is shown in the Swap class in Fig. 6. If an offer is being made, it must be posted in a new message on the endpoint before any attempt is made to react with existing offers. This ordering guarantees that there are no lost wakeups: each reagent is responsible only for those messages posted prior to it posting its own message.[4]

Once the offer (if any) is posted, tryReact peruses messages on the dual endpoint using the tail-recursive loop, tryFrom. The

[4] Our earlier work [29] with Russo on scalable join patterns gives a more detailed explanation of this protocol and its liveness properties.

loop navigates through the dual endpoint's bag using a simple cursor, which will reveal at least those messages present prior to the reagent's own message being posted to its endpoint. If a message is found, tryFrom attempts to complete the reaction; the exchange method combines the reaction and continuation of the located message with those of the reagent executing it, and actually performs the payload swap. If the reaction is successful, the final result is returned (and the result for the other reagent is separately written to its offer status). If the reaction fails, tryFrom continues to look for other messages. If no messages remain, swap behaves as if it were a disjunction: it fails permanently only if *all* messages it encountered led to permanent failures.

The code sketch we have given for channels glosses over several details of the real implementation, including avoiding fulfilling one's own offer or a single offer multiple times, and allowing multiple interactions between two reagents within a single reaction.

6. Performance

Fine-grained concurrent data structures are usually evaluated by targeted microbenchmarking, with focus on contention effects and fine-grained parallel speedup [2, 7, 9, 10, 12, 19, 20, 24]. In addition to those basic aims, we wish to evaluate (1) the extent to which reagent-based algorithms can compete with their hand-built counterparts and (2) whether reagent composition is a plausible approach for scalable atomic transfers. To this end, we designed a series of benchmarks focusing on simple lock-free collections, where overhead from reagents is easy to gauge. Each benchmark consists of n threads running a loop, where in each iteration they apply one or more atomic operations on a shared data structure and then simulate a workload by spinning for a short time. For a high contention simulation, the spinning lasts for 0.25μs on average, while for a low contention simulation, we spin for 2.5μs.

In the "PushPop" benchmark, all of the threads alternate pushing and popping data to a single, shared stack. In the "StackTransfer" benchmark, there are two shared stacks, and each thread pushes to one stack, atomically transfers an element from that stack to the other stack, and then pops an element from the second stack; the direction of movement is chosen randomly. The "EnqDeq" and "QueueTransfer" benchmarks are analogous, but work with queues instead. The stack benchmarks compare our reagent-based TreiberStack to (1) a hand-built Treiber stack, (2) a mutable stack protected by a single lock, and (3) a stack using STM. The queue benchmarks compare our reagent-based MSQueue to (1) a hand-built Michael-Scott queue, (2) a mutable queue protected by a lock, and (3) a queue using STM. For the transfer benchmarks, the hand-built data structures are dropped, since they do not support atomic transfer; for the lock-based data structures, we acquire both locks in a fixed order before performing the transfer.

We used the Multiverse STM, a sophisticated open-source implementation of Transaction Locking II [3] which is distributed as part of the Akka package for Scala.[5] Our benchmarks were run on a 3.46Ghz Intel Xeon X5677 (Westmere) with 32GB RAM and 12MB of shared L3 cache. The machine has two physical processors with four hyperthreaded cores each, for a total of 16 hardware threads. L1 and L2 caches are per-core. The software environment includes Ubuntu 10.04.3 and the Hotspot JVM 6u27.

The results are shown in Fig. 7; the x-axes show thread counts, while the y-axes show throughput (iterations/μs, so larger numbers are better). The reagent-based data structures perform universally better than the lock- or STM-based data structures. The results show that reagents can plausibly compete with hand-built concurrent data structures, while providing scalable composed operations that are rarely provided for such data structures.

[5] Although TL2 is intended for more complex data structures than the ones tested here, it is by far the most readily available production STM in the Java/Scala ecosystem.

7. Related work

7.1 Concurrent ML

Concurrent ML [22] was designed to resolve an apparent tension between abstraction and choice: if protocols are represented abstractly as functions, it is impossible to express the choice of two abstract protocols. The solution is *higher-order concurrency*, a code-as-data approach in which synchronous message-passing protocols are represented abstractly as *events*. CML's events are built up from combinators, including a choice combinator, communication combinators, and combinators for arbitrary computations not involving communication. Reagents are clearly influenced by the design of CML's events, and include variants of CML's core event combinators (communication and choice). But where CML is aimed squarely at capturing synchronous communication protocols, reagents are designed for writing and tailoring fine-grained concurrent data structures and synchronization primitives. This difference in motivation led us to include a number of additional combinators, including those dealing directly with shared state.

Originally, CML was focused on managing concurrency rather than profiting from parallelism, and this focus was reflected in its implementation. More recently, a parallel implementation of CML was proposed [23]. The key challenge is resolving uses of choice both consistently and scalably. It is addressed by *sharing*: an event making a choice is enrolled as offering a communication corresponding to each possible choice, and when a communication is accepted, the (single, shared) event is atomically marked as consumed. We follow a similar strategy in dealing with message passing, but where Parallel CML uses lock-based queues to store messages, we show how to use lock-free bags for increased parallelism (§5). We also show how to incorporate choice resolution with shared-state updates and our conjunction combinators.

7.2 Software transactional memory

Software transactional memory was originally intended "to provide a general highly concurrent method for translating sequential object implementations into non-blocking ones" [25]. This ambitious goal has led to a remarkable research literature, which has been summarized in textbook form [15]. Much of the research is devoted to achieving scalability on multiprocessors or multicores, sometimes by relaxing consistency guarantees or only providing obstruction-freedom rather than lock-freemdom [12].

Reagents, on the other hand, are aimed at a less ambitious goal: enabling the concise expression, user tailoring, and composition of fine-grained concurrent algorithms. That is, unlike STM, reagents do not attempt to provide a *universal* fine-grained concurrent algorithm. Instead, they assist in writing and using specific algorithms.

There is a clear tradeoff. Using STM, one can implement a concurrent stack or queue by simply wrapping a sequential version in an atomic block, which requires no algorithmic insight and is simpler than the stack or queue we give in §3. But even with a very clever STM, these implementations are unlikely to scale as well as our elimination stack or Michael-Scott queue; some evidence for that is shown in §6.

Reagents carve out a middle ground between completely hand-written algorithms and the completely automatic atomic blocks of STM. When used in isolation, reagents are *guaranteed* to perform only the CASes that the hand-written algorithm would, so they introduce no overhead on shared-memory operations; by recoding an algorithm use reagents, you lose nothing. Yet unlike hand-written algorithms, reagents can be composed using choice, tailored with new blocking behavior, or combined into larger atomic blocks.

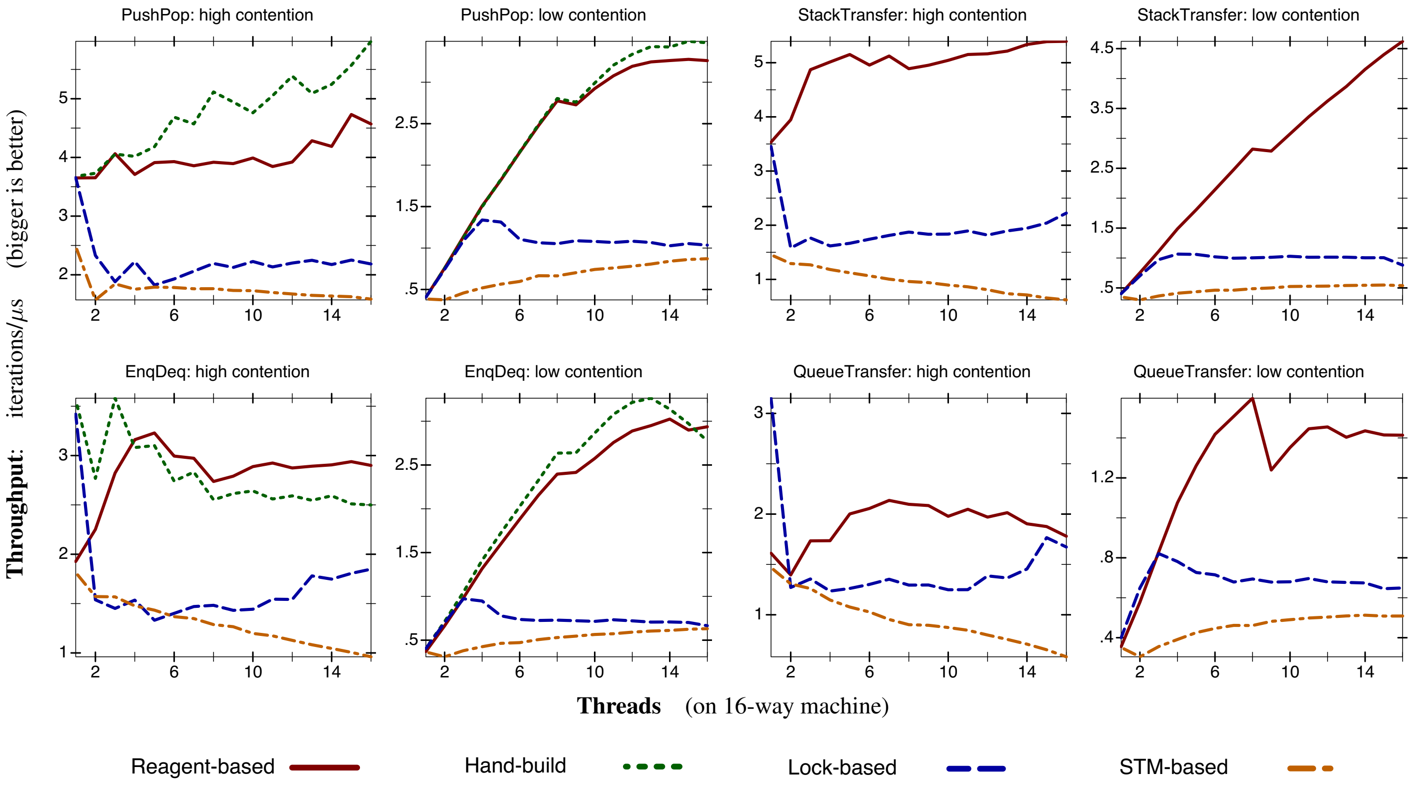

Figure 7. Benchmarking results

Haskell's STM [8] was inspirational in showing that transactions can be represented via monads [21], explicitly composed, and combined with blocking and choice operators. Reagents also form a monad, but we have chosen an interface closer to *arrows* [14], to encourage static reagent layout wherever possible (§4.1). Like orElse in Haskell's STM, our choice operator is left-biased. But unlike orElse, our choice operator will attempt the right-hand side even when the left-hand side has only failed transiently (rather than permanently).[6] While the distinction appears technical, it is crucial for supporting examples like the elimination backoff stack (§3.3).

7.3 Transactions that communicate

A central tenet of transactions is *isolation*: transactions should not be aware of concurrent changes to memory from other transactions. But sometimes it is desirable for concurrent transactions to coordinate or otherwise communicate while executing. Recent papers have proposed mechanisms for incorporating communication with STM, in the form of Communicating Transactions [16], Transaction Communicators [17], and Transactions with Isolation and Cooperation (TIC, [27]). A key question in this line of work is how the expected isolation of shared memory can safely coexist with concurrent communication. Communicating Transactions use explicit, asynchronous message passing to communicate; the mechanism is entirely separate from shared memory, which retains isolation. On the other hand, Transaction Communicators and TIC allow isolation to be weakened in a controlled way.

Our mixture of message-passing and shared-state combinators most closely resembles the work on Communicating Transactions. Of course, the most important difference is in the way we deal with shared state discussed in §7.2. We also believe that *synchronous* communication is better for expressing patterns like elimination (§3.3), since they rely on participants being mutually-aware.

There has also been work treating pure message-passing in a transactional way. Transactional Events [4] combines CML with an atomic sequencing operator. Previously, Transactional Events were implemented on top of Haskell's STM, relied on search threads[7] for matching communications, and used an STM-based representation of channels. However, Transactional Events are expressible using reagents, through the combination of swap and the conjunction combinators. Doing so yields a new implementation that does not require search threads, performs parallel matching for communication, and represents channels as lock-free bags. We are not in a position to do a head-to-head comparison, but based on the results in §6, we expect the reagent-based implementation to scale better on fine-grained workloads.

Another message-passing system with a transactional flavor is the Join Calculus [6], discussed in §3.7. The treatment of conjunction, message passing, and catalysis is essentially inherited from our previous work giving a scalable implementation of the join calculus [29].

7.4 Composing fine-grained concurrent data structures

Most of the literature on fine-grained concurrent data structures is focused on "within-object" atomicity, for example developing algorithms for inserting or removing elements into a collection atomically. However, there has been some recent work studying the problem of transferring data atomically between such fine-grained data structures [2]. The basic approach relies on a *k*CAS operation in much the same way that reagent sequencing does. The transfer methods must be defined manually, in advance, and with access to

[6] Note that retry in Haskell's STM signals a *permanent* failure, rather than an optimistic retry.

[7] The implementation can be made to work with a single search thread at the cost of lost parallelism.

the internals of the relevant data structures, whereas reagents allow arbitrary new compositions, without manual definition, and without access to the code or internals of the involved data structures. Nevertheless, the implementation of reagent composition yields an algorithm very similar to the manually-written transfer methods.

It is also possible to go in the other direction: start from STM, which provides composition, and add an "escape hatch" for writing arbitrary fine-grained concurrent algorithms within the scope of a transaction. The escape hatch can be provided through unlogged reads and/or writes to memory locations being used by transactions, as in early release [12] or elastic transactions [5]. As we discussed above (§7.2), we favor an approach where the focus is foremost on writing fine-grained algorithms, with *guarantees* about the performance and shared-memory semantics of those algorithms. Providing such guarantees via an escape hatch mechanism may be difficult or impossible, depending on the details of the STM implementation. As we showed in §3, it is also very useful to have combinators for choice, message-passing, and blocking, if one wishes to capture the full range of fine-grained concurrent algorithms.

8. Conclusion and planned work

Reagents blend together message passing and shared state concurrency to provide an abstract, compositional way of writing fine-grained concurrent algorithms. We see this work as serving the needs of two distinct groups: concurrency experts and concurrency users. Using reagents, experts can write libraries more easily, because common patterns are expressible as abstractions and many are built-in. Users can then extend, tailor and compose the resulting library without detailed knowledge of the algorithms involved.

There is significant remaining work for elucidating both the theory and practice of reagents. On the theoretical side, developing a formal operational semantics would help to clarify the interactions possible between shared state and message passing, as well as the atomicity guarantees that reagents provide. On the practical side, developing a serious concurrency library using reagents would go a long way toward demonstrating their usability. In future work, we plan to pursue both of these goals. In addition, we plan to expand the scope of reagents to include fine-grained locking as well as non-blocking data structures.

Acknowledgments

Thanks to Claudio Russo, Vincent St-Amour, Sam Tobin-Hochstadt, Jesse Tov and Mitchell Wand for insightful discussions during the development of this work. Pete Manolios provided access to benchmarking equipment. Thanks go to Neil Torronto for the development of the new Racket plot library. The anonymous reviewers provided thorough reviews which led to several improvements to the presentation. Finally, I greatly appreciate the generous financial support of Microsoft Research.

References

[1] H. Attiya and E. Hillel. Highly-concurrent multi-word synchronization. In *ICDCN*, 2008.

[2] D. Cederman and P. Tsigas. Supporting lock-free composition of concurrent data objects. In *CF*, 2010.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *DISC*, 2006.

[4] K. Donnelly and M. Fluet. Transactional events. *JFP*, 18(5 & 6):649 – 706, Oct. 2008.

[5] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, 2009.

[6] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL*, 1996.

[7] K. Fraser and T. Harris. Concurrent programming without locks. *TOCS*, 25(2), May 2007.

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, Aug. 2005.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, Jan. 2004.

[10] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003. ISBN 1581137087.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[14] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, May 2000.

[15] J. Larus and R. Rajwar. *Transactional memory*. Morgan and Claypool, 2006.

[16] M. Lesani and J. Palsberg. Communicating memory transactions. In *PPOPP*, 2011.

[17] V. Luchangco and V. Marathe. Transaction communicators: enabling cooperation among concurrent transactions. In *PPOPP*, 2011.

[18] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *SPAA*, Dec. 2003.

[19] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1):21–65, 1991.

[20] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[21] S. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL*, 1993.

[22] J. Reppy. CML: a higher order concurrent language. In *PLDI*, 1991.

[23] J. Reppy, C. Russo, and Y. Xiao. Parallel concurrent ML. In *ICFP*, Aug. 2009.

[24] W. Scherer III and M. Scott. Nonblocking concurrent data structures with condition synchronization. In *DISC*. Springer, 2004.

[25] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb. 1997.

[26] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997.

[27] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *OOPSLA*, 2007.

[28] R. Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, 1986.

[29] A. Turon and C. V. Russo. Scalable Join Patterns. In *OOPSLA*, 2011.

[30] P. Wadler. The essence of functional programming. In *POPL*, 1992.