11

Implementing reagents

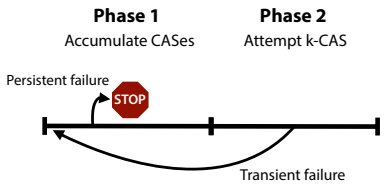
▸ Synopsis This chapter walks through the implementation of reagents (in Scala) in significant detail, which reveals the extent to which reagents turn patterns of scalable concurrency into a general algorithmic framework. It includes benchmarking results comparing multiple reagent-based collections to their hand-written counterparts, as well as to lock-based and STM-based implementations. Reagents perform universally better than the lock- and STM-based implementations, and are competitive with hand-written lock-free implementations.

"One of the joys of functional programming is the way in which apparently-exotic theory can have a direct and practical application, and the monadic story is a good example."

—Simon Peyton Jones, "Tackling the awkward squad"

11.1 Overview

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 reagents are designed to avoid this kind of overhead in the common case; it is crucial that reagents used to express scalable concurrent algorithms do not generate traffic to shared memory beyond what the algorithms require. 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 persistent failure (§10.2.1). Persistent failures indicate that the reagent cannot proceed given current conditions, and should therefore block until another thread intervenes and causes conditions to change.[1] On the other hand, a failure during the second phase, *i.e.* a failure to commit, is always a transient failure (§10.2.1). Transient failures indicate that the reagent should retry, since the reaction was halted due to active interference from another thread. In general, an in-progress reaction is represented by an instance of the `Reaction` class, and contains three lists: the CASes to be performed, the messages to be consumed,[2] and the actions to be performed after committing. A `Reaction` thus resembles the redo log used in some STM implementations.[3]

In the common case that a reagent performs only one visible (§10.5) 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

[1] *Cf.* §2.2.3.

[2] Message consumption ultimately boils down to additional CASes.

[3] Larus et al. (2006), "Transactional memory"

leads to exactly one executed CAS during reaction, with no extra overhead.[4] When a reaction encompasses multiple visible CASes or message swaps, a costlier[5] *k*CAS protocol must be used to ensure atomicity. We discuss the *k*CAS protocol in §11.4, and the common case single CAS in §11.5.1.

▸ In the implementation, `Reagent[A,B]` is an abstract class all of whose subclasses are private to the library. These private subclasses roughly correspond to the public combinator functions, which are responsible for instantiating them; each subclass instance stores the arguments given to the combinator that created it.[6]

The one combinator that does *not* have a corresponding `Reagent` subclass is sequencing `>>`. Instead, the reagent subclasses internally employ *continuation-passing style* (CPS): each reagent knows and has control over the reagents that are sequenced after it, which is useful for implementing backtracking choice.[7] Thus, instead of representing the sequencing combinator `>>` with its own class, each reagent records its own *continuation* `k`, which is another reagent. For example, while the `cas` combinator produces a reagent of type `Reagent[Unit,Unit]`, the corresponding `CAS` class has a continuation 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 trivial "halt" continuation, `Commit`, whose behavior is explained in §11.4.

Each subclass of `Reagent[A,B]` must implement its abstract methods:

```
abstract class Reagent[-A, +B] {   // the +/- are variance annotations
  def >>[C](next: Reagent[B,C]): Reagent[A,C]
  def canFail: Boolean // can this reagent fail?
  def canSync: Boolean // can this reagent send a message?
  def tryReact(a: A, rx: Reaction, offer: Offer[B]): Any
}
```

The `tryReact` method takes the input `a` (of type `A`) to the reagent and the reaction `rx` built up so far, and either:[8]

- completes the reaction, returning a result (type `B`), or
- fails, returning a failure (type `Failure`). The class `Failure` has exactly two singleton instances, `Block` and `Retry`, corresponding to persistent and transient failures respectively.

The remaining argument, `offer`, is used for synchronization and communication between reagents, which we explain next.

[4] An important implication is that the window of time between taking a snapshot of shared state and performing a CAS on it is kept small.

[5] Currently we implement *k*CAS in software, but upcoming commodity hardware is designed to support it primitively, at least for small *k*.

[6] Thus, reagents are an abstract data type whose instances are created using "smart constructors"—a very common idiom in functional programming.

[7] CPS is also needed for message passing: since a reagent will try to synchronize with any message it finds in a channel's bad, `swap` is also a form of backtracking choice.

[8] 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`.

11.2 OFFERS

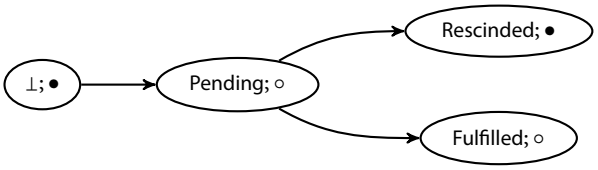
Message passing between reagents is synchronous, meaning that both reagents take part in a single, common reaction. In the implementation, this works by one reagent placing an *offer* to react in a location visible to the other.[9] The reagent making the offer either spinwaits or blocks until the offer is fulfilled;[10] if it spinwaits, it may later decide to withdraw the offer. The reagent accepting the offer sequences 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.[11] 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.

[9] *Cf.* messages in Chapter 9.

[10] It spinwaits iff it encountered a transient failure at any point during the reaction.

[11] The final answer will be the value passed the offer would have passed to its own `Commit` continuation.

Offers follow a very simple protocol (Chapter 4):



Due to the use of tokens, only the thread that originally created an offer can rescind it.

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 `null` when the reaction is first attempted; reagents make offers lazily, just as join patterns create messages lazily (§9.5.1), as we will see below.

11.3 THE ENTRY POINT: REACTING

The code for performing a reaction is given in the `!` method definition for `Reagent[A,B]`, shown in Figure 11.1. This method provides two generalized versions of the optimistic retry loops we described in Chapter 2. The retry loops are written as 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.[12] To initiate the reaction, `withoutOffer` calls the abstract `tryReact` method with

[12] *Cf.* §9.5.1.

```
def !(a: A): B = {
  val backoff = new Backoff
  def withoutOffer(): B =
    tryReact(a, empty, null) match {
      case Block ⇒ withOffer()
      case Retry ⇒
        backoff.once()
        if (canSync) 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 11.1: The `!` method, defined in `Reagent[A,B]`

the input `a`, an empty reaction to start with, and no offer. If the reaction fails in the first phase (a persistent 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 (§10.2.3) without further contention. Finally, if both phases of the reaction succeed, the final answer `ans` 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 has been made, the reagent can actually block when faced with a persistent failure; the offer will ensure that the attempted reaction is visible to other reagents, which may complete it, fulfilling the offer and waking up the blocked reagent. Blocking is performed by the `park` method provided by Java's `LockSupport` class.[13] 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.[14]

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

[13] The `park`/`unpark` methods work similarly to the signals we used in Chapter 9, but they are associated with each thread and may suffer from spurious wakeups.

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

11.4 THE EXIT POINT: COMMITTING

As mentioned in §11.2, the initial (outermost) continuation for reagents is an instance of `Commit`, which represents an "empty" reagent:

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

The emptiness of `Commit` is reflected in the first three methods it defines: it is an identity for sequencing, and it does not *introduce* any failures or synchronizations. Any failure or synchronization must be due to reagent sequenced prior to the `Commit` reagent, which always comes last.

The `tryReact` method of `Commit` makes the phase-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 *k*CAS operation: *k* compare and sets must be performed atomically. This operation, which forms the basis of STM, is in general expensive and not available in most hardware.[15] There are several software implementations that provide nonblocking progress guarantees.[16] 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 *k*CAS implementation. The advantage 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 *k*CAS protocol. Our experimental results in §2.4 indicate that even this simple *k*CAS implementation provides reasonable

[15] Though, as we have noted several times, hardware support is coming and may eventually be commonplace.

[16] Fraser and Harris (2007); Luchangco et al. (2003); Attiya and Hillel (2008)

performance—much better than STM or coarse-grained locking—but a more sophisticated *k*CAS would likely do even better.

11.5 THE COMBINATORS

11.5.1 *Shared state*

Reads are implemented by the nearly trivial `Read` class:

```
class Read[A,R](ref: Ref[A], k: Reagent[A,R])
extends Reagent[Unit,R] {
  def >>[S](next: Reagent[R,S]) = new Read[A,S](ref, k >> next)
  def canFail = k.canFail
  def canSync = k.canSync

  def tryReact(u: Unit, rx: Reaction, offer: Offer[R]) = {
    if (offer != null) ref.addOffer(offer)
    k.tryReact(ref.get(), rx, offer)
  }
}
```

A read introduces neither failures nor synchronization, but its continuation might, so `canFail` and `canSync` defer to the values in `k`. The role of reading in `tryReact` is fairly straightforward: absent an offer, we simply perform the read and pass its result to the continuation `k`, with an unchanged reaction argument `rx`. However, if an offer is present, it is recorded in a bag of offers associated with the reference (via `addOffer`). Although the read itself cannot block, the value it reads could be the proximal cause of blocking in the continuation `k`. Thus, if the continuation is preparing to block (as evidenced by the non-`null` offer), logging the offer with the read reference will ensure that the entire reagent is woken up if the reference changes. Once the offer is rescinded or fulfilled, it is considered "logically removed" from the bag of offers stored with `ref`, and will be physically removed when convenient.[17]

[17] This is essentially the same approach we used to remove messages in Chapter 9.

While the `Read` class is private to the reagent library, the corresponding `read` combinator is exported:

```
def read[A](ref: Ref[A]): Reagent[Unit, A] =
  new Read[A,A](ref, new Commit[A])
```

All of the primitive reagent combinators are defined in this style, using the `Commit` reagent as the (empty) continuation. The result type `R` of the `Read` reagent is thus initially set at `A` when reading a `Ref[A]`.

The implementation of the `cas` combinator is given by the `CAS` class, shown in Figure 11.2. Its `tryReact` method is fairly simple, but it illustrates a key optimization we have mentioned several times: if the reaction so far has no CASes, and the continuation is guaranteed to succeed, then the entire reagent

```
class CAS[A,R](ref: Ref[A], ov: A, nv: A, k: Reagent[Unit,R])
extends Reagent[Unit,R] {
  def >>[S](next: Reagent[R,S]) = new CAS[A,S](ref, ov, nv, k >> next)
  def canFail = true
  def canSync = k.canSync

  def tryReact(u: Unit, rx: Reaction, offer: Offer[R]) =
    if (!rx.hasCAS && !k.canFail)  // can we commit immediately?
      if (ref.cas(ov, nv))         // try to commit
        k.tryReact((), rx, offer)  // successful; k MUST succeed
      else Retry
    else // otherwise must record CAS to reaction log, commit in k
      k.tryReact((), rx.withCAS(ref, ov, nv), offer)
}
```

Figure 11.2: The `CAS` class

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 *k*CAS 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 *k*CAS, then the current `cas` is recorded into a new `Reaction` object,[18] which is passed to the continuation `k`. In either case, the continuation is invoked with the unit value as its argument.

11.5.2 *Message passing*

We represent each endpoint of a channel as a lock-free bag.[19] 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.[20]

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

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

[18] The `withCAS` method performs a *functional update*, *i.e.,* returns a new `Reaction` object. It is important not to mutate the reaction objects: reagents use backtracking choice (§11.5.3), and at various points in the branches of such a choice reaction objects may be used to advertise synchronizations (§11.5.2).

[19] It is possible to build the bag itself using non-blocking reagents, thereby bootstrapping the library.

[20] The tradeoffs here are essentially the same as in Chapter 9.

it returns a value of type `R`, which matches the final answer type of the sender's offer.

The core implementation of `swap` is shown in the `Swap` class in Figure 11.3. 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.[21] On the other hand, if there is no offer, `Swap` attempts to complete by consuming a message on the dual endpoint without ever creating (or publishing) its own message—exactly like the lazy message creation of §9.5.1.

[21] The design rationale and key safety/liveness properties here are exactly the same as those in Chapter 9.

Once the offer (if any) is posted, `tryReact` peruses messages on the dual endpoint using the tail-recursive loop, `tryFrom`. The 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 dual message is found, `tryFrom` attempts to complete a reaction involving it. To do this, it must *merge* the in-progress reaction of the dual message with its own in-progress reaction:

- The `++` operation on a pair of `Reaction` produces a new reaction with all of their CASes and post-commit actions.
- The `SwapK` inner class is used to construct a new continuation for the dual message. This new continuation uses the `withFulfill` method of `Reaction` to record a fulfillment[22] of the dual message's offer with the final result of the reagent in which that dual message was embedded.
- When `SwapK` is invoked as part of a reaction, it invokes the original continuation `k` with the payload of the dual message.
- If the reaction is successful, the final result is returned (and the result for the other reagent is separately written to its offer status). Recall that, in this case, the `Commit` reagent will first *rescind* the `offer` of `Swap.tryReact`, if any. Thus, if `Swap` had earlier advertised itself through its own message, it removes that advertisement before instead consuming an advertised message on the dual endpoint.[23] Just as in Chapter 9 consuming a message logically removed it, here rescinding or fulfilling the offer associated with a message logically removes the message from the bag.
- 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 persistently only if *all* messages it encountered led to persistent failures. The failure logic here closely resembles that of the `retry` flag in §9.3.

[22] Fulfillment includes waking the reagent if it is blocked on the offer.

[23] `Swap` may fail to rescind its message, but only if some other thread has fulfilled its offer; in this case, `Commit` aborts the attempt to consume a message on the dual channel and simply returns the result from the fulfilling thread.

```
class Swap[A,B,R](ep: Endpoint[A,B], k: Reagent[B, R])
extends Reagent[A,R] {
  def >>[S](next: Reagent[R,S]) = new Swap[A,S](ep, k >> next)
  def canFail = true
  def canSync = true

  // NB: this code glosses over some important details
  //     discussed in the text
  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) =>
          val merged =
            msg.senderK               // complete sender's continuation
            >> new SwapK(msg.payload, // then complete our continuation
                         msg.offer)
          merged.tryReact(a, rx ++ msg.senderRx, 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
  }

  // lift our continuation to a continuation for the dual sender
  class SwapK[S](dualPayload: B, dualOffer: Offer[S])
  extends Reagent[S,R] {
    def >>[T](next: Reagent[R,T]) = throw Impossible // unreachable
    def canFail = true
    def canSync = k.canSync

    def tryReact(s: S, rx: Reaction, myOffer: Offer[S]) = {
      k.tryReact(dualPayload, rx.withFulfill(dualOffer, s), myOffer)
    }
  }
}
```

Figure 11.3: The `Swap` class

The code we have given for `Swap` glosses over some corner cases that a full implementation must deal with. For example, it is possible for a reagent to attempt to `swap` on both sides of a channel, but it should avoid fulfilling its own offer in this case. Similarly, if a reagent `swaps` on the same channel multiple times, the implementation should avoid trying to consume the same message on that channel multiple times.

11.5.3 *Disjunction: choice*

The implementation of choice is pleasantly simple:

```
class Choice[A,B](r1: Reagent[A,B], r2: Reagent[A,B])
  extends Reagent[A,B] {
  def >>[C](next: Reagent[B,C]) =
    new Choice[A,C](r1 >> next, r2 >> next)
  def canFail = r1.canFail || r2.canFail
  def canSync = r1.canSync || r2.canSync

  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
    }
}
```

Choice attempts a reaction with either of its arms, trying them in left to right order. As explained in §10.2.3, a persistent failure of choice can only result from a persistent failure of *both* arms.[24] The right arm is tried even if the left arm has only failed transiently.

[24] The accumulation of the "`Retry`" signal here is reminiscent of the `retry` flag in §9.3.

11.5.4 *Conjunction: pairing and sequencing*

To implement the pairing combinator $*$, we first implement combinators `first` and `second` that lift reagents into product types; see Figure 11.4. These combinators are associated with *arrows*[25] in Haskell, and are useful for building up complex wiring diagrams.

[25] Hughes (2000), "Generalising monads to arrows"

```
def first[A,B,C] (r: Reagent[A,B]): Reagent[A × C, B × C] =
  new First(r, new Commit[B × C])
def second[A,B,C](r: Reagent[A,B]): Reagent[C × A, C × B] =
  new Second(r, new Commit[C × B])
```

```
class First[A,B,C,R](r: Reagent[A,B], k: Reagent[B × C,R])
extends Reagent[A × C,R] {
  def >>[S](next: Reagent[R,S]) = new First[A,B,C,S](r, k >> next)
  def canFail = r.canFail || k.canFail
  def canSync = r.canSync || k.canSync
  def tryReact(both: A × C, rx: Reaction, offer: Offer[R]) =
    (r >> Lift(b ⇒ (b, both._2)) >> k).tryReact(both._1, rx, offer)
}
// Second is defined symmetrically
```

Figure 11.4: Arrow-style lifting into product types

With them in hand, we can define a `pair` combinator[26] quite easily. The $*$ method on reagents is just an alias for the `pair` combinator, to support infix syntax.

[26] This combinator would be called &&&, or "fanout", in Haskell's arrow terminology.

```
def pair[A,B,C](r1: Reagent[A,B], r2: Reagent[A,C]): Reagent[A,B × C] =
  lift(a ⇒ (a, a)) >> first(r1) >> second(r2)
```

11.5.5 *Computational reagents*

The `lift` combinator, defined in Figure 11.5 by the `Lift` class, is the simplest reagent: it blocks when the function to lift is undefined, and otherwise applies the function and passes the result to its continuation.

```
class Lift[A,B,R](f: A ⇀ B, k: Reagent[B,R])
  extends Reagent[A,R] {
  def >>[S](next: Reagent[R,S]) = new Lift[A,B,S](c, k >> next)
  def canFail = k.canFail
  def canSync = k.canSync
  def tryReact(a: A, rx: Reaction, offer: Offer[R]) =
    if (f.isDefinedAt(a)) k.tryReact(f(a), rx, offer)
    else Block
}
```

Figure 11.5: The `Lift` class

The implementation of computed reagents (Figure 11.6) is exactly as described in §10.5: 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 persistent failure. The implementation makes clear that the reads and writes performed within the computation `c` are invisible: they do not even have access to the `Reaction` object, and so they cannot enlarge the atomic update performed when it is committed.

```
class Computed[A,B,R](c: A ⇀ Reagent[Unit,B], k: Reagent[B,R])
  extends Reagent[A,R] {
  def >>[S](next: Reagent[R,S]) = new Computed[A,B,S](c, k >> next)
  def canFail = true
  def canSync = true
  def tryReact(a: A, rx: Reaction, offer: Offer[R]) =
    if (c.isDefinedAt(a)) (c(a) >> k).tryReact((), rx, offer)
    else Block
}
```

Figure 11.6: The `Computed` class

11.6 CATALYSIS

Thus far, our survey of the reagent implementation has focused wholly on reactants. What about catalysts?

It turns out that very little needs to be done to add support for the `dissolve` operation. Catalysts are introduced by invoking `tryReact` with an instance of a special subclass of `Offer`. This "catalyzing" subclass treats "fulfillment" as a no-op—and because it is never considered fulfilled, the catalyst is never used up. Because fulfillment is a no-op, multiple threads can react with the catalyst in parallel.

11.7 BENCHMARK RESULTS

11.7.1 *Methodology and experimental setup*

As we mentioned in Chapter 9, scalable concurrent data structures are usually evaluated by targetted microbenchmarking, with focus on contention effects and fine-grained parallel speedup.[27] 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.

[27] Mellor-Crummey and Scott 1991; Michael and Scott 1996; Herlihy et al. 2003; Scherer, III and Scott 2004; Hendler, Shavit, et al. 2004; Fraser and Harris 2007; Cederman and Tsigas 2010; Hendler, Incze, et al. 2010

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

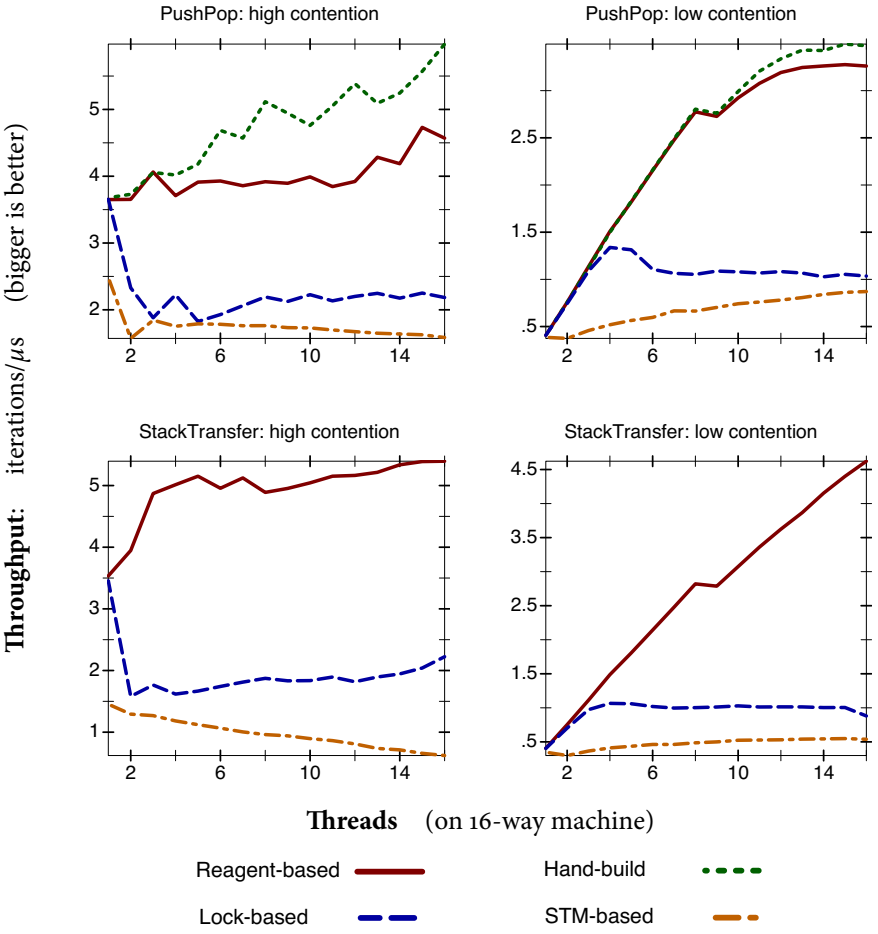


Figure 11.7: Benchmark results for stacks

an element from the second stack; the direction of movement is chosen randomly. 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 "EnqDeq" and "QueueTransfer" benchmarks are analogous, but work with queues instead. 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 Dice et al. 2006 which is distributed as part of the Akka package for Scala. 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.

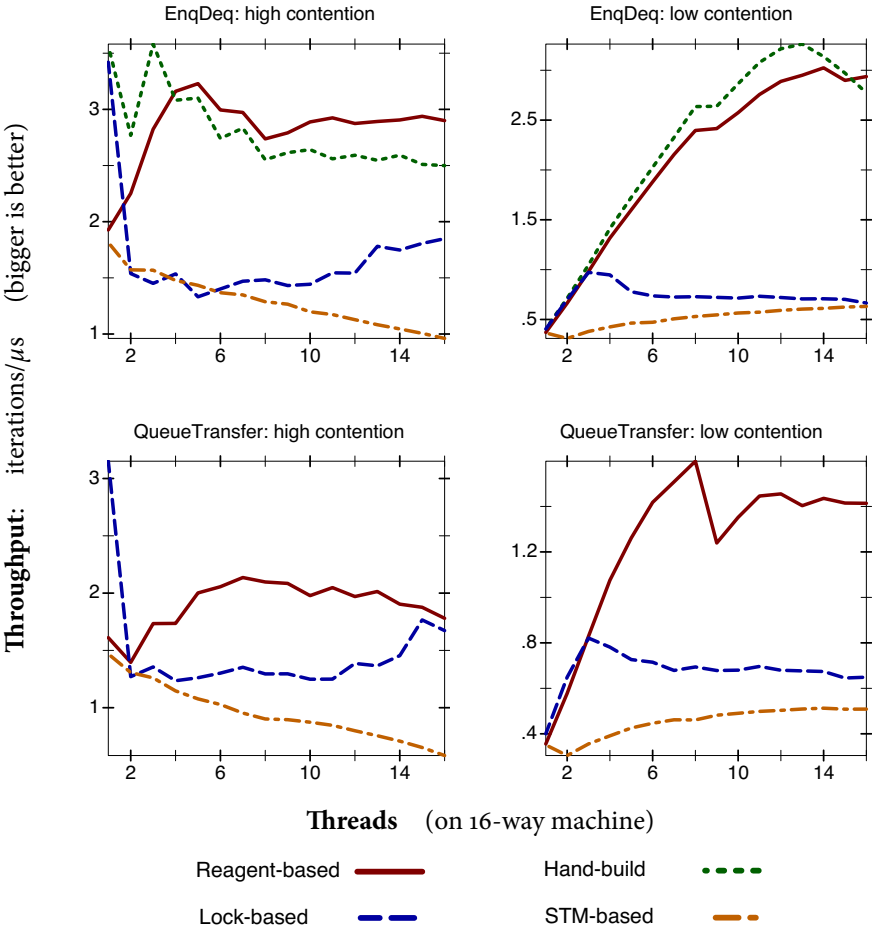


Figure 11.8: Benchmark results for queues

11.7.2 *Results*

The results are shown in Figures 11.7 and 11.8; the x-axes show thread counts, while the y-axes show throughput (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 competete with hand-built concurrent data structures, while providing scalable composed operations that are rarely provided for such data structures.

References

Attiya, Hagit and Eshcar Hillel (2008).
Highly-concurrent multi-word synchronization. *In proceedings of Distributed Computing and Networking (ICDCN)*. Springer Berlin Heidelberg, pages 112–123 (cited on page 169).

Cederman, Daniel and Philippas Tsigas (2010).
Supporting lock-free composition of concurrent data objects. *In proceedings of the ACM International Conference on Computing Frontiers (CF)*. New York, New York, USA: ACM Press, pages 53–62 (cited on page 176).

Dice, Dave, Ori Shalev, and Nir Shavit (2006).
Transactional locking II. *In proceedings of Distributed Computing (DISC)*() (cited on page 177).

Fraser, Keir and Tim Harris (2007).
Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2) (cited on pages 169, 176).

Hendler, Danny, Itai Incze, Nir Shavit, and Moran Tzafrir (2010).
Flat combining and the synchronization-parallelism tradeoff. *In proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, New York, USA: ACM Press, pages 355–364 (cited on page 176).

Hendler, Danny, Nir Shavit, and Lena Yerushalmi (2004).
A scalable lock-free stack algorithm. *In proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, New York, USA: ACM Press, pages 206–215 (cited on page 176).

Herlihy, Maurice, Victor Luchangco, Mark Moir, and W.N. N Scherer, III (2003).
Software transactional memory for dynamic-sized data structures. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)* (cited on page 176).

Hughes, J (2000).
Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111 (cited on page 174).

Larus, J.R., R. Rajwar, and Transactional Memory (2006).
Transactional memory. Morgan and Claypool (cited on page 165).

Luchangco, Victor, Mark Moir, and Nir Shavit (2003).
Nonblocking k-compare-single-swap. *In proceedings of the ACM Sympo-*

sium on Parallelism in Algorithms and Architectures (SPAA) (cited on page 169).

Mellor-Crummey, John M. and Michael L. Scott (1991).
Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65 (cited on page 176).

Michael, Maged M. and Michael L. Scott (1996).
Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *In proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, pages 267–275 (cited on page 176).

Peyton Jones, Simon (2001).
Tackling the awkward squad. *Engineering theories of software construction*, pages 47–96. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/ (cited on page 165).

Scherer, III, William N. and Michael L. Scott (2004).
Nonblocking Concurrent Data Structures with Condition Synchronization. *In proceedings of Distributed Computing (DISC)*. Springer, pages 174–187 (cited on page 176).