Lock-free programming for the masses

KC Sivaramakrishnan
University of Cambridge

Théo Laurent
ENS de Lyon

Efficient concurrent programming libraries are essential for taking advantage of fine-grained parallelism on multicore hardware. We present reagents*, a composable, lock-free concurrency library for expressing fine-grained parallel programs on Multicore OCaml. Reagents offer a high-level DSL for experts to specify efficient concurrency libraries, but also allows the consumers of the libraries to extend them further without knowing the details of the underlying implementation.*

1 Motivation

Designing and implementing scalable concurrency libraries is an enormous undertaking. Decades of research and industrial effort has led to state-of-the-art concurrency libraries such as `java.util.concurrent` (JUC) for the JVM and `System.Collections.Concurrent` (SCC) for the .NET framework. These libraries are often written by experts and have subtle invariants, which makes them hard to maintain and improve. Moreover, it is hard for the library user to safely combine multiple atomic operations. For example, while JUC and SCC provide atomic operations on stacks and queues, such atomic operations cannot be combined into larger atomic operations. On the other hand software transactional memory (STM) offers composability, but STM based data structures are generally less efficient than their lock-free counterparts.

Turon et al. [6] introduced reagents, an expressive and composable library which retains the performance and scalability of lock-free programming. Reagents allow isolated atomic updates to shared state, as well as message passing communication over channels. Furthermore, reagents provide a set of combinators for sequential composition à la STM, parallel composition à la Join calculus [2], and selective communication à la Concurrent ML [4], while being lock-free.

2 Combinators

The basic reagents combinators are presented in Figure 1. A reagent value with type `('a,'b)t` represents an atomic transaction that takes an input of type `'a` and returns a value of type `'b`. The basic atomic operations are exchanging message on an endpoint of a channel through `swap` and updating a shared reference through `upd`. The `swap` operation blocks the calling thread until a matching swap operation is available on the dual endpoint.

The atomic reference update operation `upd`, takes a function which is applied to the current value of the reference (of type `'a`) and the input value (of type `'b`), and is expected to return an optional pair of the new value for the reference and a return value (of type `'c`). Importantly, if the update function returns `None`, then the invoking thread blocks until the reference is updated. Reagent implementation takes care of the blocking and signalling necessary for thread wake up.

```
type ('a,'b) t

(* channel communication *)
val swap : ('a,'b) endpoint -> ('a,'b) reagent

(* shared memory *)
val upd : 'a ref -> ('a -> 'b -> ('a * 'c) option)
          -> ('b,'c) reagent

(* composition *)
val (>>>) : ('a,'b) t -> ('b,'c) t -> ('a,'c) t
val (<*>) : ('a,'b) t -> ('a,'c) t -> ('a,'b * 'c) t
val (<+>) : ('a,'b) t -> ('a,'b) t -> ('a,'b) t
```

Figure 1: Basic reagents combinators

The most important feature of reagents is that it allows composition of reagent transactions in sequence `>>>` and in parallel `<*>`, and also to selectively choose one of the available operations `<+>`. Importantly, these combinators being arrows [3], enable important optimisations that cover the common case and help reagents achieve performance commensurate to hand-written implementations. Reagents library also exposes monadic combinators for convenience, at the cost of forgoing optimisation opportunities.

3 Treiber stack

The following is a reagent implementation of the Treiber lock-free stack [5].

```
module R = Reagent

module Treiber_stack : sig
  type 'a t
  val create  : unit -> 'a t
  val push    : 'a t -> ('a, unit) R.t
  val pop     : 'a t -> (unit, 'a) R.t
  val try_pop : 'a t -> (unit, 'a option) R.t
end = struct
  type 'a t = 'a list R.ref

  let create () = R.ref []

  let push r x =
    R.upd r (fun xs x -> Some (x::xs,()))

  let try_pop r = R.upd r (fun l () ->
    match l with
```

```
    | [] -> Some ([], None)
    | x::xs -> Some (xs, Some x))

  let pop r = Ref.upd r (fun l () ->
    match l with
    | [] -> None
    | x::xs -> Some (xs,x))
end
```

We utilise a shared reference of type `'a list ref` to represent the stack and use the `upd` operation to perform atomic operations on the stack. The important take away from this snippet is that the code is no more complicated than a sequential stack implementation. he logic for backoff, retry, blocking and signalling are hidden behind the reagents implementation. In particular, the `pop` operation blocks the calling thread until the stack is non-empty. TThus, the experts can write efficient concurrency libraries using reagents while preserving readability (and as a consequence maintainability) of code.

Furthermore, since the stack interface is exposed as reagents, the individual operations can be further composed. For example, given two Treiber stacks `s1` and `s2`, `pop s1 >>> push s2` transfers elements atomically between the stacks, `pop s1 <*> pop s2` consumes elements atomically from both of the stacks, and `pop s1 <+> pop s2` consumes an element from either of the stacks. Importantly, the composition preserves the optimisations and blocking/signalling behaviours. Thus, reagents allow users of the library to arbitrarily combine and extend the functionality without knowing about the underlying implementation.

4 Implementation

The key idea behind the implementation is that the reagent transaction executes in two phases. The first phase involves collecting all the compare-and-swap (CAS) operations necessary for the transaction, and the second phase is invoking a k-CAS operation (emulated in software). The failure to gather all the available CASes constitutes a permanent failure, causing the thread to explore other alternatives in the case of a selective communication or block otherwise. The failure in the second phase means that there is active interference from other concurrent threads, in which case the transaction is retried.

Performance of the Reagents depends critically on having fine-grained control over threads and schedulers for implementing backoff loops, blocking and signalling. However, one of the main ideas of multicore OCaml is not to bake in the thread scheduler into the compiler but rather describe them as libraries. To this end, the reagents library is functorized over the following generic scheduler interface:

```
module type Scheduler = sig
  type 'a cont (* continuation *)
  effect Suspend : ('a cont -> 'a option) -> 'a
  effect Resume  : 'a cont * 'a -> unit
end
```

The interface itself only describes the scheduler's effects, whose behaviour is defined by the handlers [1]. `perform (Suspend f)` applies `f` to the current continuation, and allows the Reagent library to stash the thread on the unavailable resource's wait queue. The return type of `f` is an option to handle the case when the resource might have become available while suspending. If `f` returns `None`, then the control returns to the scheduler. Once the resource becomes available, the reagent library performs the `Resume` effect to resume the suspended thread.

Using the reagents library, we have implemented a collection of composable concurrent data and synchronization structures such as stacks, queues, sets, hash tables, countdown latches, reader-writer locks, condition variables, exchangers, atomic counters, etc. [1]

5 Limitations and Future Work

Reagents are less expressive than STM which provides serializability. But in return, Reagents provide stronger progress guarantee (lock-freedom) over STM (obstruction-freedom). A reagent transaction operating more than once on the same memory location will fail at runtime. Abstractly, this behaviour is disallowed since it cannot be represented as a k-CAS operation. Due to this restriction, the transaction `pop s1 >>> push s1` always fails, and prohibits important patterns such as atomically pushing or popping multiple values from the same stack. We are extending the original reagent semantics to relax this invariant. As a result, reagents will have snapshot isolation semantics. While this is weaker than serializability semantics offered by the STM, we will retain the benefit of lock-freedom.

References

[1] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Users and Developers Workshop*, 2015.

[2] C. Fournet and G. Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In *Applied Semantics*. 2002.

[3] J. Hughes. Programming with arrows. In *Advanced Functional Programming*. 2004.

[4] J. Reppy, C. V. Russo, and Y. Xiao. Parallel Concurrent ML. In *ICFP*, 2009.

[5] R. K. Treiber. *Systems programming: Coping with parallelism*. IBM Almaden Research Center, 1986.

[6] A. Turon. Reagents: Expressing and Composing Fine-grained Concurrency. In *PLDI*, 2012.

[1] https://github.com/ocamllabs/reagents