SPLG Project Report E2012

Michael-Scott Queue in Racket

Florian Biermann – fbie@itu.dk

December 9, 2012

This project report for "Programming Languages Seminar", autumn 2012 at ITU, is concerned with the difficulties of implementing the Michael-Scott Queue in Racket and Typed Racket. The report shows the implementations in detail, talks about issues in typing the code and outlines weaknesses in a software CAS approach.

Contents

1. Introduction and Motivation **3**

2. Implementation **3**
2.1. Compare-And-Swap 3
2.2. Michael-Scott Queue 5
2.3. Michael-Scott Queue Variant 6
2.4. Typed Version 6

3. Discussion **7**

4. Conclusion and Future Work **8**

A. Source Code **9**

1. Introduction and Motivation

Over the course of Programming Languages Seminar, many different advanced topics in current computer science research have been introduced. The focus of each technology is varying greatly. For a final project, it is tempting to try to combine a number of these and see, how they perform together.

This project is concerned with implementing a Michael-Scott Queue [1] in Racket. The topics being combined here are gradual typing and non-blocking concurrency. This is especially interesting, as Racket does not feature a dedicated compare-and-swap (CAS) operation executed in hardware. Therefore, the first step is to show the implementation of a software CAS in Racket.

The remainder of this report is structured as follows. First, I will give a detailed description of the implementation, outlining differences between a Java-like CAS operation and the new CAS-like operation I implemented for Racket. Further, I will describe how the Michael-Scott Queue implementation is designed and finally concern about gradually typing this implementation.

For evaluation, I will show how the Racket implementation of the Michael-Scott queue performs in comparison. The report ends with a conclusion, discussing these results and outlining some future work.

2. Implementation

In this section, I will give a detailed overview of the implementation of each data structure I implemented over the course of the project.

2.1. Compare-And-Swap

To implement a CAS operation without access to hardware instructions, it is necessary to wrap the data, on which CAS should be performed, into a container. This container will then be assigned a semaphore that guards access to the CAS function. When a thread executes CAS, no other thread may invoke CAS on the same container in parallel. If it does, its execution will be blocked.

The container is implemented in the polymorphic struct *atomic* as illustrated in Fig. 1. The struct holds a mutable field *ref* that can be manipulated safely through CAS (Fig. 1, ll. 11-22).

atomic is constructed through a second polymorphic function *make-atomic* which accepts any type as parameter. *make-atomic* internally passes a newly created semaphore from the Racket standard library to the *atomic*, together with the value. The semaphore reference is not mutable and is therefore consistent for the lifetime of any *atomic*, making CAS save.

Each time CAS is executed on an instance of *atomic*, it calls an internal method, that performs the actual CAS with *call-with-semaphore*, supplying the semaphore field of the reference. *call-with-semaphore* is a shorthand for waiting and taking the lock on the semaphore, then execute code, and finally release the lock again.

```
#lang racket

(require racket/future)

(struct atomic (sem fsem [ref #:mutable]))

(define (make-atomic ref)
  (atomic (make-semaphore 1) (make-fsemaphore 1) ref)
  )

(define (CAS ref exp new)
  (define (cas)
    (define success #t)
    (if (equal? (atomic-ref ref) exp)
      (set-atomic-ref! ref new)
      (set! success #f))
      success
    )
  (fsemaphore-wait (atomic-fsem ref))
  (let ([res (call-with-semaphore (atomic-sem ref) cas)])
    (fsemaphore-post (atomic-fsem ref))
    res)
  )

(provide
 (contract-out
  [atomic-ref (-> atomic? any/c)]
  [make-atomic (-> any/c atomic?)]
  [CAS (-> atomic? any/c any/c boolean?)]))
```

Figure 1: Untyped code for *atomic*.

Since Racket distinguishes between scheduled Threading and true parallelism, another kind of semaphore needs to be added. *fsemaphore* is a parallel-safe semaphore that locks the execution of "futures" [1], which is the Racket implementation of parallel threads. This semaphore is assigned to the field *fsem* on *atomic* and its locking mechanism encloses the traditional semaphore. This ensures that *atomic* is safe for single- and multi-processor parallelism.

The only publicly provided functions from this implementation are

- *make-ref* as a safe constructor for *atomic*
- standard getter function *atomic-ref* that returns the *ref* value on the *atomic*
- CAS as the only operation that allows to manipulate the value of *ref* on *atomic*

```
1
2  (define (dequeue q)
3    (define return (void))
4    (while #t
5           (let* [(head (msq-head q))
6                  (tail (msq-tail q))
7                  (next (atomic-ref (node-next (atomic-ref head))))]
8             (when (equal? head (msq-head q))
9               (if (equal? (atomic-ref head) (atomic-ref tail))
10                  (if (equal? next (void))
11                      (break)
12                      (CAS (msq-tail q) tail next))
13                  (when (CAS (msq-head q) (atomic-ref head) next)
14                    ((set! return (node-value next))
15                     (break))
16                    )))))
17   return
18   )
```

Figure 2: Untyped implementation of DEQUEUE of the Michael-Scott Queue.

2.2. Michael-Scott Queue

The implementation of the non-blocking Michael-Scott queue [1] is based on the re-vised pseudo code of their implementation [2], translated into Racket.

The implementation consists of two structs: *node*, which holds the actual values and is interlinked with other instances of its type, and *msq*, the container for the linked list that represents the queue. A constructor function *make-msq* is used to initialize the linked list correctly, i.e. instantiate the next pointer of *node* with an *atomic*. By using Racket as implementation language, the

As an alteration, DEQUEUE returns the value of the item that has been dequeued (see Fig. 2, ll.2, 13, 16). This makes the data structure a more practical for actual usage.

Using a *while*-construct instead of a recursive function makes it easier to repeat the step in case that the queue is not consistent at any point during ENQUEUE or DEQUEUE. If it was implemented only recursively, for each inconsistency, that is checked in an *if*-expression without an *else*-clause, the recursive function would have to be called again. This would be hard to read and therefore the *while*-loop is the better choice even in a functional language.

It is clearly visible that, due to the implementation of *atomic*, for wich CAS is defined, the original implementation of Michael and Scott [1] can be translated directly into Racket. This is especially true for ENQUEUE, as there is no need to alter the implementation to make it more useful, opposing DEQUEUE (see Fig. 3 for the implementation).

[1] See http://docs.racket-lang.org/reference/futures.html for further details on futures in Racket.
[2] See http://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html#nbq for the re-vised pseudo code of [1]

```
(define (enqueue value q)
  (while #t
         (let* [(node (node value (make-atomic (void))))
                (tail (msq-tail q))
                (next (atomic-ref (node-next (atomic-ref tail))))]
           (when (equal? tail (msq-tail q))
             (when (equal? next (void))
               (if (CAS (node-next (atomic-ref tail)) next node)
                   ((CAS (msq-tail q) (atomic-ref tail) node)
                    (break))
                   (CAS (msq-tail q) tail next))))))
  (void)
  )
```

Figure 3: Untyped implementation of ENQUEUE of the Michael-Scott Queue.

2.3. Michael-Scott Queue Variant

This variant of the Michael-Scott Queue was introduced by Turon et al. [3]. The main difference is, that there is no tail pointer that can lag behind. The tail is determined by recursing to the end of the queue. This is a valid approach, as at no time the *next* reference of a node is altered after it was changed from *void* to an actual succeeding node, even if it got dequeued. This is elaborated further in [3].

The implementation is purely functional, as the pseudo-code of the variant itself suggests, not using any kind of while-construct but only recursive calls. This is possible as there is no tail in this variant which could be lagging behind, making updating the tail a redundant task. The implementation is illustrated in Fig 4.

While this is easier to code, the performance cost for finding the tail in each step is great in comparison to the original proposed implementation.

2.4. Typed Version

For the sake of staying within the scope of this project, only the re-vised Michael-Scott Queue was translated into typed code.

The first observation is that *while* is not compatible with Typed Racket and the argument of code readability does not hold anymore. Another issue with this is however that the current implementation of ENQUEUE is now using way more space, in regard to how many attempts to enqueue fail, as some calls to CAS need to be bound to local variables in order to be able to execute them before a succeeding recursive call to *try* (see Fig. 5 for the actual implementation).

The second observation is that, as the *msq* struct is not initialized with any value but *void*, it cannot be polymorphic but it will hold values of type *Any*. This could be changed by allowing an initial value for the queue, which would be set to the tail position of the queue while head remains a dummy *node*. Also, this makes the typing easier, though lacking possibly vital information.

```
(define (enqueue value q)
  (let ([n (node value (make-atomic (void)))])
    (define (try c)
      (if (equal? (atomic-ref (node-next c)) (void))
          (when (not (CAS (node-next c) (void) n))
              (try c))
          (try (atomic-ref (node-next c))))
      )
      (try (atomic-ref (msq-head q))))
  )

(define (dequeue q)
  (define (try)
    (let ([n (atomic-ref (msq-head q))])
      (if (equal? (atomic-ref (node-next n)) void)
          void
          (if (CAS (msq-head q) n (atomic-ref (node-next n)))
              (node-value (atomic-ref (node-next n)))
              (try)))))
  (try)
  )
```

Figure 4: Untyped implementation of the Michael-Scott Queue variant by Turon et al.

3. Discussion

According to Swaine et al. [2], programms written in Typed Racket have advantages over programms written in plain Racket, when it comes to parallelism. When a parallel program is compiled, the Racket compiler determines the safety of the code for parallel execution. If the program does not fulfill all requirements, a so called "slow path" will be triggered [2], leading eventually towards sequential execution with the additional overhead paid for managing parallel processes on top.

Swaine et al. claim that typing the program in Typed Racket counters this behavior, as they also demonstrate in [2]. It makes therefore sense to measure the amount of parallel work and compare the typed and the untyped implementation of the Michael-Scott Queue.

Fig. 6 shows how the untyped implementation does not perform work in parallel at all. Instead, each thread pauses entirely as soon as another thread touches the queue.

However, the current typed implementation of the Michael-Scott Queue does not trigger a "fast path" in the Racket compiler. The performance visualizer still shows 750 "Blocking" items for a test with four parallel processes where each enqueues 250 randomly chosen integers into the same Michael-Scott Queue instance.

It is likely, that the compiler detects the use of *fsemaphore* objects, which guard the CAS on *atomic* (see Sec. 2.1). Since the Racket compiler regards the main thread not as a parallel thread (even though it is), it is reasonable to assume that the 1000 occurrences of *fsemaphore* minus the share of the main thread give the 750 "Blocking" items as shown by the visualizer.

```
(: enqueue (Any MS-Queue -> Void))
(define (enqueue value q)
  (: try ( -> Void))
  (define (try)
    (let* [(node (node value (make-atomic (void))))
           (tail (cast (atomic-ref (msq-tail q)) Node))
           (next (atomic-ref (node-next tail)))]
      (if (equal? tail (atomic-ref (msq-tail q)))
        (if (equal? next (void))
          (if (CAS (node-next tail) next node)
              (let ([a (CAS (msq-tail q) tail node)])
                (void))
              (let ([a (CAS (msq-tail q) tail next)])
                (try)))
          (try))
        (try))))
  (try)
  )
```

Figure 5: Typed implementation of ENQUEUE.

It is also possible that this behavior is due to lacking type information, as suggested in Sec. 2.4. As Typed Racket and Racket are still undergoing many changes and are topic of research interest, it was not possible to gain much information on these corner cases during the duration of this project.

Other tweaks to the code to make the compiler choose a "fast path" as described by Swaine et al. [2] are unfortunately not adaptable to our case: since they deal with fast fourier transformation, their approaches target arithmetic operations on and memory allocation of numbers. Since the Michael-Scott Queue is polymorphic (and in the typed case, it is even broader than that, since its underlying type is *Any*), it is not possible to adapt their proposed changed to the current implementation.

4. Conclusion and Future Work

Over the course of this report, we saw that it is very well possible to write a generally working implementation of the Michael-Scott Queuein Racket as well as Typed Racket. It became, however, obvious, that true parallelism in Racket is not as trivial to achieve, because of compiler features that are designed to stop the programmer from executing possibly error prone code in parallel.

As the reason for this is supposedly the frequent use of semaphores to ensure correctness on CAS operations, it may be a good idea to add CAS instead as a built-in function to the Racket compiler or as a C++ extension to the standard library. This could then circumvent triggering "slow paths" in the execution of parallel programs in Racket and therefore provide programmers with the possibility to write programmes with fine-grained parallelism.

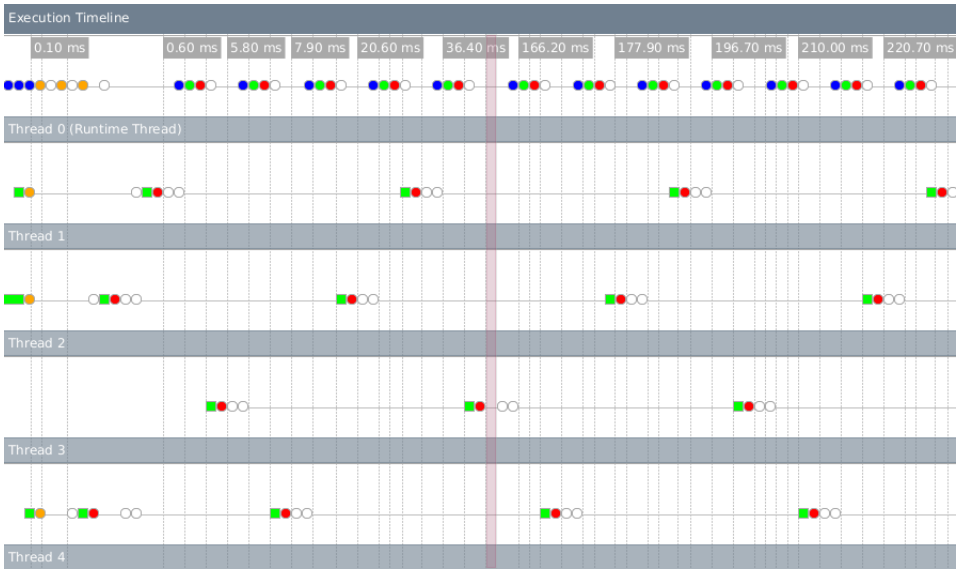

Figure 6: Screenshot from the Future Visualizer showing the execution timeline for the untyped Michael-Scott Queue implementation. Blue points indicate a blocking operation.

It is still possible, that parallel execution is only prevented by too broad type annotations. Since the project duration is limited, this is not featured here, but considered for future work.

References

[1] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, page 267–275, New York, NY, USA, 1996. ACM.

[2] J. Swaine, B. Fetscher, V. St-Amour, R. B. Findler, and M. Flatt. Seeing the futures: profiling shared-memory parallel racket. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, page 73–82, 2012.

[3] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. *Submitted for publication*, 2012.

A. Source Code

All implemented source code regarding this project, including unit-tests for classical and parallel threading, can be found at `https://github.com/fbie/racket-cas`.