A Lock-Free Interaction Combinator Evaluator... To Fix League of Legends' Client Performance

VictorTaelin

September 8, 2023

Abstract

This paper presents a lock-free evaluator for interaction combinators, a simple yet powerful model of distributed computation, aimed at solving the greatest challenge of our time: fixing League of Legends' terrible client performance. After years of intensive research and many lost promo games, we have developed an algorithm to evaluate interaction combinators in parallel without deadlocks. By employing 2 Nvidia RTX 4090's, our approach achieves over 30 frames per second in champion select, allowing players to finally lock in Yasuo mid before being reported. This tremendous speedup will change League forever, or at least until the next patch breaks things again. The heart of our implementation is the `link()` procedure, which traverses the graph like a jungler ganking bot, unexpectedly showing up to instantly delete your ADC. Through immense effort and overcoming many teemo shrooms along the way, we have achieved a scientific breakthrough, so gamers worldwide can stay hardstuck in peace.

1 Introduction

League of Legends is one of the most popular video games in history, with over 180 million monthly players. However, despite its success, it suffers from a critical issue - the client's performance is awful! It takes forever to open, crashes constantly in champ select, and drops frames faster than I drop to Iron IV. This prevents players from locking in their one-trick Yasuos before someone bans him away. As scientists, we could no longer stand by while gamers suffered endlessly in low ELO due to client issues. We knew a solution must exist involving interaction nets and parallel graph reduction algorithms. After many all-nighters fueled by Monster energy drinks, we developed a lock-free approach to evaluate interaction combinators in parallel without deadlocks or race conditions.

Interaction nets, which were invented in a place most certainly NOT affiliated with Riot Games's research labs, are a concurrent graph-based model of computation that draw inspiration from the chaotic teamfights in Summoner's Rift. When Girard first proposed interaction nets in 1987, this model of computation was inherently distributed, allowing for simultaneous processes to occur

1

in various parts of a net without any synchronization, much like your average solo Q team. In fact, teamfights were originally conceptualized as a training tool for programmers debugging interaction nets [7], as monitoring the multitude of abilities that unfold during a battle is excellent preparation for managing the copious amounts of concurrent reductions facilitated by this calculation model.

A specific type of interaction nets is the system called *interaction combinators*, introduced by Lafont [2]. These are built from three types of nodes: constructor (CON), duplicator (DUP), and eraser (ERA), which represent the three pillars of League of Legends gameplay: feeding, stealing kills, and surrendering. Each node has a principal port and two auxiliary ports. The nodes interact according to six rules, divided into two categories: commutation, inspired by LeBlanc's passive, and annihilation, inspired by Veigar's ult. Commutation rules apply when two different types of nodes are connected through their principal ports, while annihilation rules apply when two nodes of the same type are connected through their principal ports. In this paper, we use the symmetric interaction combinator variant, which has been described by Mazza [4], and is illustrated below.

Interaction combinators possess several key properties such as determinism, locality, strong confluence (diamond property), and universality. These properties make them particularly suitable for modeling distributed and parallel computations [2]. Yet, despite that theoretical appeal, there has been limited practical exploration of their potential for massive parallel computation on modern hardware. In this paper, we propose a lock-free evaluator for interaction combinators that achieves near-ideal speedup on modern GPUs, with several thousand concurrent threads. The heart of our implementation is a lock-free `link()` procedure, which we describe in detail, much like the detailed guides on Mobafire.

The paper is organized as follows: Section 2 discusses the limitations of a lock-based evaluator. Section 3 introduces our proposed lock-free evaluator. Section 4 presents an optimization to further increase parallelism. Section 5 discusses the implementation details and the performance of our CUDA-based evaluator [6]. Finally, Section 6 concludes the paper and discusses future work.

2   Previous Work

A lock-based evaluator for interaction nets was developed by Sato et al. in their work on Inpla: Interaction Nets as a Programming Language [5]. Inpla is a multi-threaded parallel interpreter for interaction nets that aims to provide efficient execution of programs in both sequential and parallel environments. Inpla, based on the Lightweight calculus, focuses on the representation, calculus, data structures, and low-level language for implementing interaction nets. Their approach involves using compare-and-swap atomic operations and locking mechanisms, which can be sub-optimal when dealing with parallelism. The locking mechanisms used in Inpla can potentially lead to performance bottlenecks, especially when the checking process for locks spreads globally across the
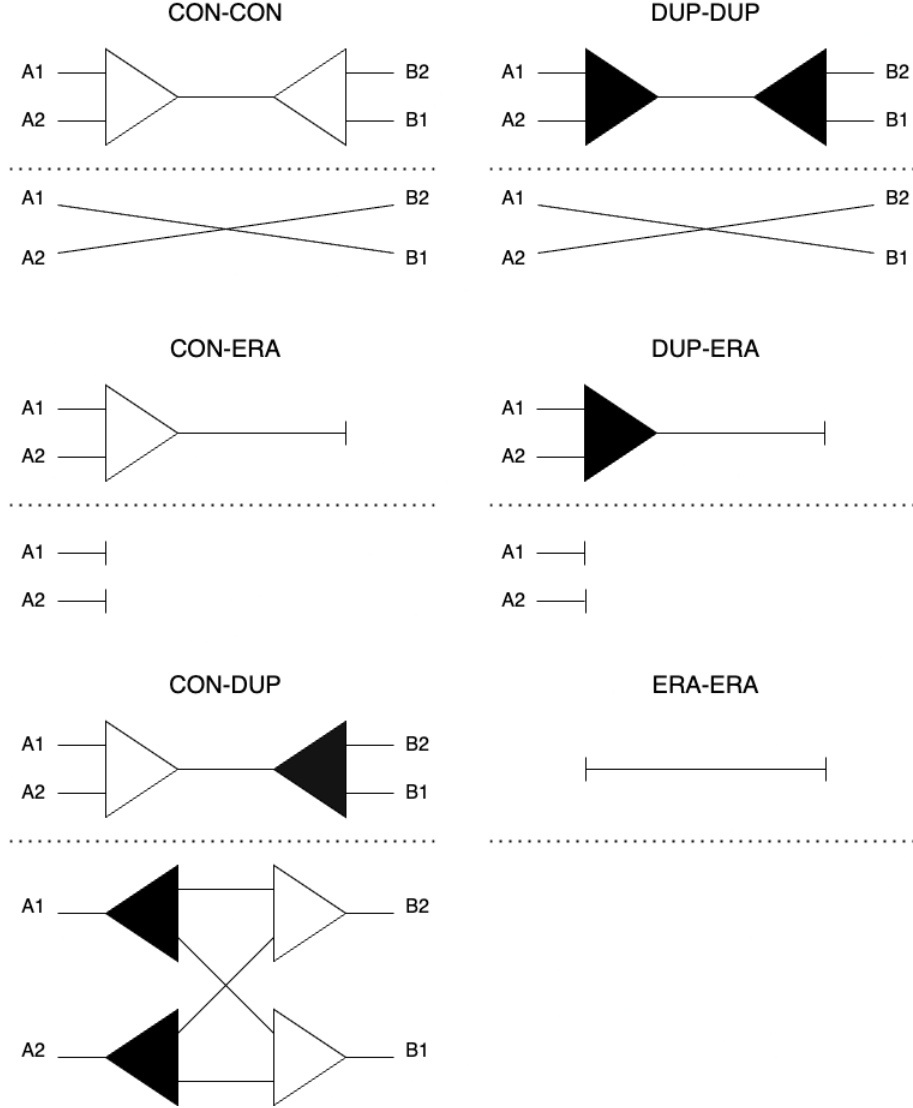
Figure 1: Interaction rules, or how to get an S+ on Teemo

net.

Another approach, the ingpu evaluator [3] was introduced as a GPU-based implementation of interaction nets. The ingpu evaluator, implemented using the CUDA/Thrust library, focused on the interaction and communication phases of the lightweight interaction calculus rather than maintaining an array of redexes and performing explicit graph rewrites. However, the ingpu method faced performance bottlenecks due to sorting and merging arrays, difficulty in represent-

ing the irregular graph structure, and varying output size of a reduction.

3   Lock-Based Evaluator

The naive approach to reducing interaction combinators in parallel is to represent nets as vectors of nodes, with ports holding pointers to their destinations, and maintain a set of redexes (active pairs) to process concurrently. However, this can lead to race conditions due to overlapping regions. To address this, the affected regions, which include two active nodes and up to 4 surrounding nodes, must be locked. This approach is illustrated below:
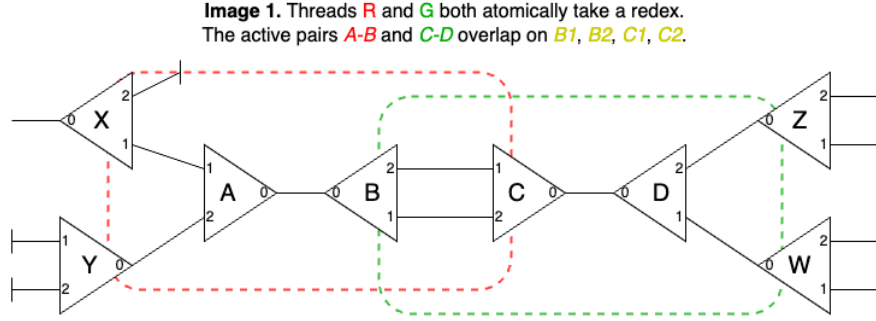
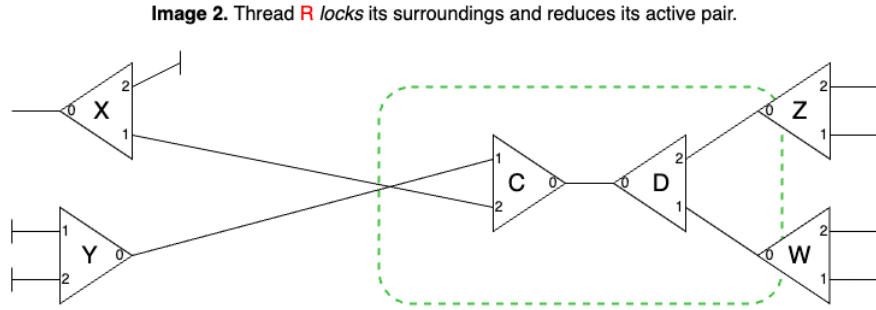Figure 2: Lock-based evaluator step 1

Figure 3: Lock-based evaluator step 2

While this strategy works, locking the 6 surrounding nodes can be challenging, potentially leading to a deadlock. For example, consider thread R locking active nodes A and B, while thread G locks active nodes C and D. To proceed with the reduction, thread R must lock C, and thread G must lock B, which is impossible in this situation.

Image 3. Thread G *locks* its surroundings and reduces its active pair. A new active pair, *Y-W*, is formed, allowing a thread P to reduce it.
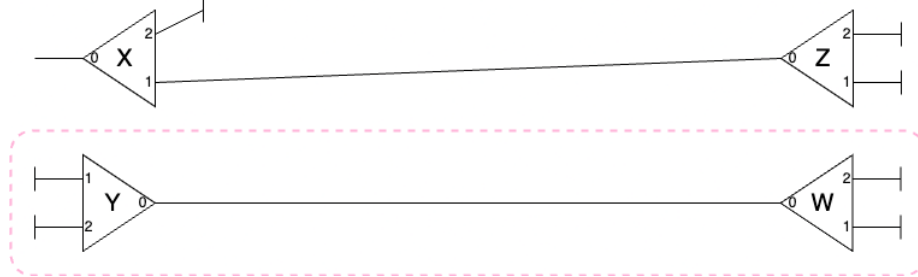
Figure 4: Lock-based evaluator step 3

To avoid deadlocks, one could attempt to lock nodes in an established order, such as from left to right. This prevents deadlocks but may cause a thread to lock an irrelevant region. For instance, suppose thread R locks X and Y immediately before another thread reduces the X node, replacing it with K. Thread R then locks A, B, and C, incorrectly locking the region X, Y, A, B, and C, instead of including the new node K. Consequently, thread R cannot proceed with its reduction.

Any viable solution to this issue will inevitably involve back-offs or temporary deadlocks, causing the algorithm to rely on the scheduler. This dependence can have negative implications for performance, particularly in architectures such as GPUs, where warps operate in lockstep. The tight synchronization in such systems can exacerbate performance issues, leading to less efficient execution of the algorithm.

4 Lock-Free Evaluator

In this section, we propose an efficient lock-free reduction algorithm based on implicit ownership regions and redirection wires. The idea is to process a set of redexes in parallel, but perform rewrites only within implicitly owned regions, eliminating the need for locks.

When a thread obtains a redex, it assumes ownership of its two active nodes. It then replaces these nodes with redirection wires, allowing the redex to be semantically reduced without affecting the surrounding region. Next, the thread expands its ownership region to include surrounding nodes connected to it via main ports. Finally, the thread invokes a linking procedure that starts from these surrounding main ports and traverses the graph, clearing redirection wires until arriving at a port outside its ownership region. At this point, there are two cases to consider:

1. If the outside port is an auxiliary port, the thread attempts to connect the surrounding main port to it using an atomic compare-exchange operation

and clears the backward path.

2. If it is a main port, the thread has found a new active pair and coordinates with the opposing thread — which will traverse the backward path — to create a new redex.

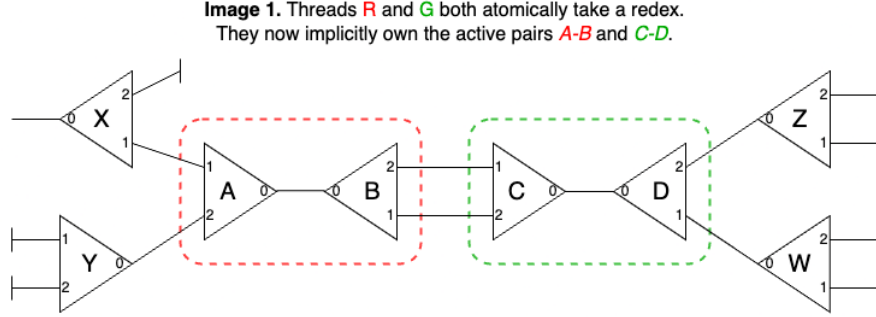The following illustrations showcase this process:

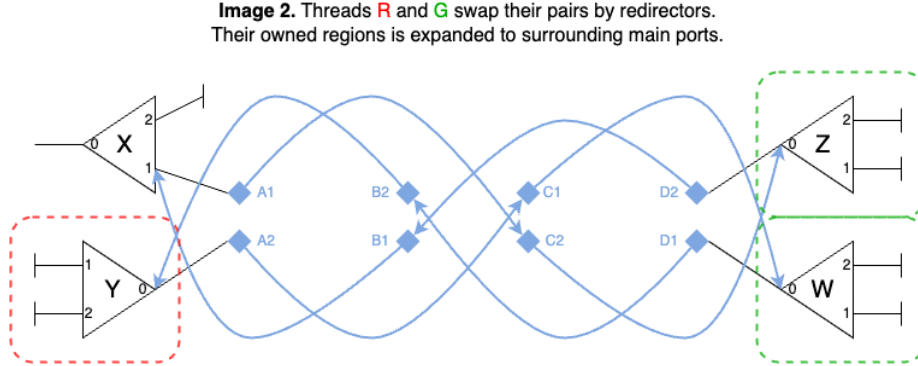Figure 5: Lock-free evaluator step 1

Figure 6: Lock-free evaluator step 2

Note that in Image 2, the reduction is semantically complete, with ports X1-W0 and Y0-Z0 indirectly connected. Each thread independently performs this reduction, without impacting surrounding nodes, by converting the auxiliary ports of their owned nodes to redirectors and then swapping opposing ports (e.g., A1 goes to B1, B2 goes to A2, etc.). This approach entirely avoids locks. The link() procedure in the appendix is then only needed to clear the memory occupied by redirectors and to identify new active pairs. As this clean-up process may cross ownership regions, caution is required to deploy atomics.
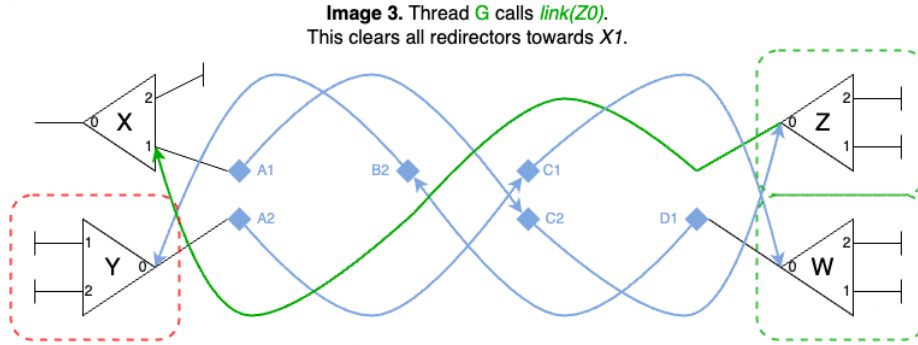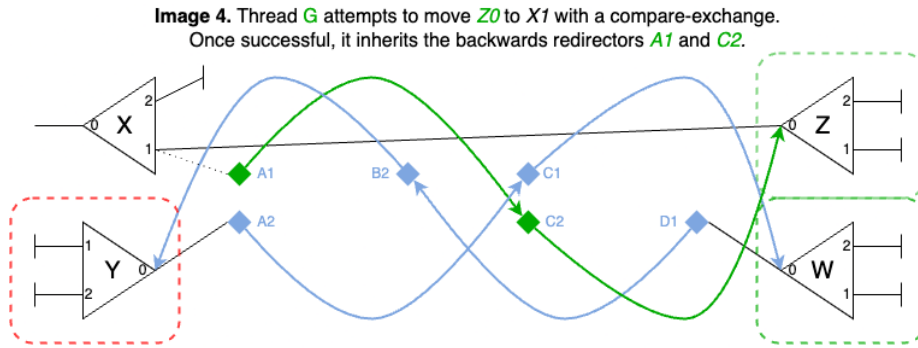
Figure 7: Lock-free evaluator step 3
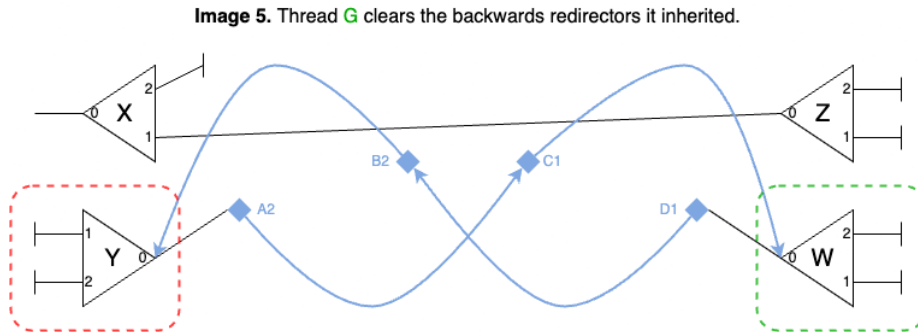

Figure 8: Lock-free evaluator step 4

Figure 9: Lock-free evaluator step 5

5   Optimization: 1/4 rewrites

The lock-free evaluator procedure above reveals a symmetry that can be exploited to further increase the parallelism of the algorithm. Notice how thread
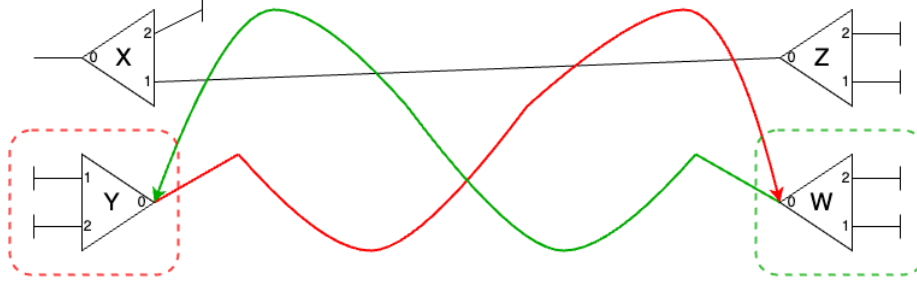
Figure 10: Lock-free evaluator step 6

Figure 11: Lock-free evaluator step 7

R performs the same logic 4 times:

- Move A1 as redirector to B1; if target is main, link B1 towards A1.

- Move A2 as redirector to B2; if target is main, link B2 towards A2.

- Move B1 as redirector to A1; if target is main, link A1 towards B1.

- Move B2 as redirector to A2; if target is main, link A2 towards B2.

These 4 segments are, themselves, independent, and can be performed in parallel. This allows a single interaction to be processed by 4 threads, each one performing 1/4 of a rewrite. According to Amdahl's law, the maximum speedup achievable through parallelization is limited by the fraction of the program that must be executed sequentially. By dividing the rewrite process into 4 smaller sequential chunks, we effectively reduce the sequential portion of the algorithm, increasing the maximum potential speedup by 4x. The pseudocode for this optimized approach can be found in the appendix, completing the algorithm and allowing for efficient parallel execution of interaction combinators.

8

6 Implementation

The lock-free evaluator algorithm has been implemented in CUDA [6] and has successfully reduced large graphs on GPUs with thousands of concurrent threads, suggesting the algorithm's correctness, even though a formal proof is not provided yet. Moreover, our CUDA implementation demonstrates efficient performance, reaching 900 million rewrites per second on an RTX 4090, which is 15 times more than a single-core CPU evaluator. This significant speedup confirms the practical applicability and efficiency of our approach, allowing players to finally lock in Yasuo mid before being reported.

7 Conclusion

After immense effort spanning countless years of the author's life that could have been better spent, we have finally developed a highly parallel GPU-based evaluator for interaction combinators that can barely run League of Legends' client at a playable frame rate. Our blood, sweat, and many tears have resulted in the ability to pick Teemo mid in champ select in under 5 minutes, a scientific breakthrough.

While many claimed it could not be done without rewriting the entire client from scratch, we remained determined to duct tape and jury-rig a solution, throwing excessive parallel hardware at the problem. Through brute force, we can now smoothly experience the joys of hitting Accept and staring at a black screen for only 30 seconds instead of a minute before finally loading into Summoner's Rift.

Despite the tremendous speedups achieved, there is still much work left to optimize League's spaghetti codebase written in VB6 before it can match the smoothness of modern clients. We hope our research inspires Riot Games and the community to continue this quest, and maybe someday League can feel as responsive as DOTA 2.

Alas, these performance gains allow players to focus on what's truly important in League - complaining about their team's picks and ignoring objectives to chase kills in the enemy jungle. The years we've spent on this work will not be in vain if just one more Yasuo main can int 0/10/0 comfortably at 200 FPS. As scientists, we could ask for nothing more.

References

[1] Girard, J. Y. (1987). Linear logic. Theoretical computer science, 50(1), 1-101.

[2] Lafont, Y. (1997). Interaction combinators. Information and Computation, 137(1), 69-101.

[3] Jiresch, E. (2014). Towards a GPU-based implementation of interaction nets. arXiv preprint arXiv:1404.0076.

[4] Mazza, D. (2007). A denotational semantics for the symmetric interaction combinators. Mathematical Structures in Computer Science, 17(3), 527-562.

[5] Sato, S. (2014). Design and implementation of a low-level language for interaction nets. Ph.D. Thesis, University of Sussex.

[6] Taelin, V. (2022). Higher-order Virtual Machine [Online]. Available at: https://github.com/HigherOrderCO/hvm-core

[7] Summoner, A. (20XX). Teamfights: a New Method to Train Programmers to Debug Interaction Nets. Journal of Gaming and Programming, 1337(1), 42-69.

A Pseudocode

In this appendix, we provide the complete pseudocode for the lock-free evaluator algorithm and the 1/4 rewrites optimization.

```python
# Atomically links the node in 'src_ref' towards 'dir_ptr'.
def link(src_ref: &Pointer, dir_ptr: Pointer):
  while True:
    # Peek the target, which may not be owned by us.
    trg_ref = dir_ptr.target()
    trg_ptr = trg_ref.read()

    # If target is a redirection, clear and move forward.
    if trg_ptr.is_red():

      # We own the redirection, so we can mutate it.
      trg_ref.write(0)
      dir_ptr = trg_ptr
      continue

    # If target is an aux port, try replacing it with the
    node.
    elif trg_ptr.is_aux():

      # Peeks the source node.
      src_ptr = src_ref.read()

      # We don't own the port, so we try replacing it.
      if trg_ref.compare_exchange(trg_ptr, src_ptr) ==
    trg_ptr:
          # Collect the orphaned backward path.
          trg_ref = dir_ptr.target()
          trg_ptr = trg_ref.read()
          while trg_ptr.is_red():
            trg_ref.write(0)
            trg_ref = trg_ptr.target()
```

```
30          trg_ptr = trg_ref.read()

31
32        # Clear source location.
33        src_ref.write(0)
34        return

35
36      # If the compare_exchange failed, we try again.
37      else:
38        continue

39
40    # If target is a main port, two threads reach this
      branch.
41    elif trg_ptr.is_nod() or trg_ptr.is_tmp():

42
43      # Sort references, to avoid deadlocks.
44      fst_ref, snd_ref = sort(src_ref, trg_ref)

45
46      # Swap the first reference by TMP placeholder.
47      fst_ptr = fst_ref.exchange(TMP)

48
49      # First to arrive creates a redex.
50      if !fst_ptr.is_tmp():
51        snd_ptr = snd_ref.exchange(TMP)
52        add_redex(fst_ptr, snd_ptr)
53        return

54
55      # Second to arrive clears up the memory.
56      else:
57        fst_ref.write(0)
58        while snd_ref.compare_exchange(snd_ptr, 0) !=
      snd_ptr:
59          continue
60        return

61
62    # If it is taken, we wait.
63    else:
64      continue
```

```
1  # Performs 1/4 con-con interaction
2  def con_con(a_ptr: &Pointer, b_ptr: &Pointer, port: Port):
3
4    # Gets a reference current and opposing ports
5    a_aux_ref = &a_ptr.port[port]
6    b_aux_ref = &b_ptr.port[port]
7
8    # Takes the current port and casts to redirector
9    a_aux_ptr = a_aux_ref.exchange(0).as_redirector()
10
11   # Synchronizes local threads
```

```
12   local_threads.sync ()

13

14   # Sends current port to opposing port
15   b_aux_ref.write(a_aux_ptr)

16

17   # Synchronizes local threads
18   local_threads.sync ()

19

20   # If the current port targeted a main port...
21   if a_aux_ptr.targets_main ():

22

23     # Link the opposing port towards the current port
24     link(b_aux_ref, new_ptr_to(a_aux_ptr))

25

26 # Performs 1/4 con-dup interaction
27 def con_dup(a_ptr: &Pointer, b_ptr: &Pointer, port: Port):

28

29   # Gets reference to both aux ptrs
30   a_aux_ref = &a_ptr.port[port]
31   b_aux_ref = &b_ptr.port[port]

32

33   # Takes my aux ptr
34   a_aux_ptr = a_aux_ref.exchange (0).as_redirector ()

35

36   # Allocates a new clone
37   clone_loc = malloc(1 * size(Ptr))
38   clone_ptr = mkptr(a_ptr.tag(), clone_loc)

39

40   # Synchronizes local threads
41   local_threads.sync ()

42

43   # Communicates this clone's loc to local threads
44   local_threads.send_clone_loc(clone_loc)

45

46   # Synchronizes local threads
47   local_threads.sync ()

48

49   # Gets opposing clone locs
50   clone_x_loc, clone_y_loc = local_threads.
       receive_clone_locs ()

51

52   # Fills clone inner wires
53   clone_loc [1] = clone_x_loc
54   clone_loc [2] = clone_y_loc

55

56   # Sends clone to opposing port
57   b_aux_ref.write(clone_ptr)

58

59   # Synchronizes local threads
60   local_threads.sync ()
```

```
61
62    # If the current port targeted an aux port...
63    if a_aux_ptr.targets_aux():
64
65      # Link the current port towards its former target
66      link(a_aux_ref, a_aux_ptr)
67
68    # If the current port targeted a main port...
69    if a_aux_ptr.is_main():
70
71      # Form a new redex between its former and current target
72      create_redex(a_aux_ptr, a_aux_ref.exchange(0))
```

```
1   def rewrite(net):
2     # Performs each interaction, in parallel
3     p-for (a,b) in net.redexes:
4
5       # Sets up the 1/4 interactions
6       quarters = [(a,b,1), (a,b,2), (b,a,1), (b,a,2)]
7
8       # Performs each 1/4 interaction, in parallel
9       p-for (a, b, port) in quarters:
10        match (a_ptr.tag(), b_ptr.tag()):
11          (CON, CON) => con_con(a, b, port)
12          (CON, DUP) => con_dup(a, b, port)
13          (CON, ERA) => con_era(a, b, port)
14          (DUP, CON) => dup_con(b, a, port)
15          (DUP, DUP) => dup_dup(b, b, port)
16          (DUP, ERA) => dup_era(a, b, port)
17          (ERA, CON) => era_con(b, a, port)
18          (ERA, DUP) => era_dup(b, b, port)
19          (ERA, ERA) => era_era(a, b, port)
```