

# Relativistic Programming in Haskell

## Using Types to Enforce a Critical Section Discipline

Ted Cooper

Portland State University  
theod@pdx.edu

Jonathan Walpole

Portland State University  
walpole@pdx.edu

### Abstract

*Relativistic programming (RP)* is a concurrent programming model in which readers can safely access shared data concurrently with writers that are modifying it. RP has been used extensively in operating system kernels, most notably in Linux, and is widely credited with improving the performance and scalability of Linux on highly concurrent architectures. However, using RP primitives safely is tricky, and existing implementations of RP do not identify most unsafe uses statically. This work introduces Monadic RP, a GHC Haskell library for RP, and presents an example of its use. To our knowledge, Monadic RP is the first RP implementation in Haskell. It provides a novel mechanism to statically rule out a subset of unsafe relativistic programs by using types to separate read-side and write-side critical sections, and to restrict the operations available in each. This work describes the current status of our implementation, presents a brief experimental evaluation of it, and discusses directions for ongoing and future work.

**Categories and Subject Descriptors** D.1.3 [Concurrent Programming]

**Keywords** relativistic programming, read-copy update, monads, type safety

### 1. Introduction

*Relativistic Programming (RP)* [29] is a concurrent programming model where a *writer* thread can modify a linked shared data structure (such as a linked list or a tree) safely while other threads (*readers*) concurrently traverse or query it. Unlike conventional lock-based models, in which the writer acquires a lock to exclude concurrent readers (and vice versa), or transactional memory models, where concurrent reads and writes to the same memory location conflict and cause aborts and retries [8], RP allows both the writer and concurrent readers who access the same data to proceed and complete successfully. So RP offers greater concurrency between writers and readers, but because of this concurrency, readers may not observe concurrent writes in the same order writers issue them. Also, readers may not agree on the order of concurrent writes. That is, when there is concurrent reading, the order of writes is not

absolute, but instead depends on the rate, path, and direction of each reader’s traversal through the data structure relative to the writer’s updates<sup>1</sup>. In general, each of these traversals is a *read-side critical section*, a sequence of related reads from shared data with an explicitly defined beginning and end. At the beginning of each read-side critical section, readers can access a shared data structure only through some reference all threads share, such as the head of a list or the root of a tree.

The core of a relativistic programming implementation is a set of ordering mechanisms, that is, ways for writers to guarantee that readers agree on the order of writes. One such primitive, unique to RP [22], we will call `synchronizeRP`<sup>2</sup>. A writer can guarantee that the states two writes create are *observed* by all readers in order by invoking `synchronizeRP` between them. This primitive introduces a delay that will not terminate until all readers that are active at the start of the delay have finished their current read-side critical sections. The RP literature calls this delay a *grace period*. Note that `synchronizeRP` does not prevent readers from starting new queries or traversals *during* the delay, or wait for these readers to complete these new critical sections. Each of these readers began after the first write was committed, and so cannot observe the value it replaced. Now the writer can issue the second write, certain that any reader who observes it had an opportunity to see the first. Another way of saying this is that, by waiting for a grace period to complete, the writer obtains a guarantee that the first write *happens before* [16] the second. If readers and writers follow a discipline of rules including those we introduce in Section 2.2, then readers can safely traverse or query a data structure updated by writers at any time, so they avoid the overhead of synchronizing (for instance, acquiring a lock) before reading shared data.

Researchers have developed relativistic implementations of common data structures — including linked lists [4], hash tables [30], and red-black trees [11] — bringing concurrency between readers and writers, and extremely low-cost read-side primitives, to programs that use these data structures. RP is used widely in the Linux kernel, and was introduced in the form of the Linux *Read-Copy Update (RCU)* API [23], whose read-side critical sections execute deterministically (without blocking or retrying) and run almost as fast as unsynchronized sequential code. For highly concurrent, read-mostly workloads that allow relaxed data structure semantics, such as software routing tables [23] and SELinux policy rules [25], these read-side characteristics improve performance, reliability, and scalability. For these reasons, RCU is now used widely in all subsystems of the Linux kernel [24], often to replace *reader-writer locking*. De-

Copyright ©2015 Ted Cooper and Jonathan Walpole. This work is licensed under the Creative Commons Attribution 4.0 International license. To view a copy of the license, visit <http://creativecommons.org/licenses/by/4.0/legalcode>.

<sup>1</sup> Triplett et al. [29] chose the name “relativistic programming” to emphasize this partial ordering of writes, which is akin to *relativity of simultaneity* in physics, described by Einstein [6].

<sup>2</sup> We color write operations `red`, and will introduce additional colors for other kinds of operations.

spite RP's benefits and deployment footprint however, there is little existing language- or tool-based support for it. Instead, programmers are responsible for knowing and manually following the RP discipline. The work presented here takes some first steps towards addressing this gap.

Specifically, we introduce *Monadic RP*, which, to our knowledge, is the first implementation of RP in Haskell. This work is inspired by the GHC *Software Transactional Memory (STM)* implementation of Harris et al. [8], and by the user-level RCU C library (`urcu`) of Desnoyers et al. [4]. Like GHC STM, Monadic RP uses features unique to the *Glasgow Haskell Compiler (GHC)* and uses a monadic abstraction to constrain shared data and the concurrent computations that operate on it. Unlike GHC STM, Monadic RP does not enforce an atomicity property among sequences of reads and writes, nor does it enforce a total ordering of writes to the same data. Monadic RP allows concurrent reading and writing of the same data (something that is not possible if strongly atomic transactions are used for all reads and writes), and it provides mechanisms that writers can use as needed to guarantee that other threads see concurrent writes in order. Hence, Monadic RP presents a programming model based on *causal ordering*. There is good reason (and evidence) to believe that these two programming models are complementary. For example, Howard and Walpole [10] modified an STM implementation to support relativistic reading while retaining transactional concurrency control among writers.

The main advantage Haskell brings to RP is that its type system captures the structure of imperative computations in enough detail to enforce on programmers a unique discipline tailored to each type of computation. Such a discipline can, for example, reject programs that call functions inside computations where they don't belong. Monadic RP uses types to mark explicitly and separate read-side and write-side critical sections, and to guarantee that readers do not block or write to shared memory. In ongoing work, we are extending Monadic RP to statically check reader traversal order and writer update order to guarantee that readers observe causally related updates in order.

This work presents a new implementation of RP, situates it in the context of existing implementations, and explains and demonstrates some of its properties:

- Section 2.1 explains the causal ordering guarantees RP relies on using two examples, and outlines ongoing work to statically check that relativistic programs obey a discipline that provides these guarantees. In turn, Section 5 demonstrates that when these examples are constructed using Monadic RP, they execute correctly because the implementation provides the causal ordering guarantees they need.
- Section 2.2 elaborates rules from the discipline for correct RP.
- Section 3:
  - situates Monadic RP in the context of existing work on RP correctness (detailed in Section 6),
  - motivates and describes the abstractions Monadic RP provides and how they enforce rules from Section 2.2,
  - presents an implementation of a read-side critical section for the examples from Section 2.1 that uses these abstractions,
  - describes a case where another RP implementation allows violations of rules Monadic RP enforces, and
  - outlines ongoing work on Monadic RP to enforce more rules.
- Section 4 describes the implementation of Monadic RP, filling in details about the mechanisms that enforce rules from Section 2.2.
- Section 5:

- presents implementations of the write-side critical sections for the examples from Section 2.1, and
- presents a brief experimental evaluation based on these examples that tests whether the causal ordering mechanisms Monadic RP provides are necessary and sufficient.
- Section 7 summarizes this work and directions for future work.

This paper describes future work inline, rather than in a separate section, in an effort to show directly how it connects to the work we present.

## 2. Background

### 2.1 Causal Ordering

#### 2.1.1 Grace periods enforce causal ordering

Most implementations of RP are in systems that rely on the programmer to include code that explicitly allocates and reclaims memory, so the programmer must write that code to use grace periods to guarantee that writers do not reclaim memory that readers may yet visit. In this case, a writer begins a grace period after replacing one or more nodes in a data structure accessible through a shared reference by updating the references in existing nodes to point to new copies. At this point, one or more readers may have already loaded the addresses of old copies, whose fields must remain available for the readers' inspection. Once the grace period ends, each reader has finished its current read-side critical section, and old copies of updated nodes and unlinked nodes are no longer reachable from the shared reference, so they can be safely deallocated. That is, the writer initiates a grace period and waits until the readers announce they have each finished a read-side critical section, and so cannot access unlinked nodes. Through this channel of communication, the writer learns from the readers when it is safe to reclaim memory.

This memory reclamation guarantee is an instance of a more general property: Unlinking a node and reclaiming its memory are causally ordered operations. That is, reclaiming is a consequence of unlinking, so when a writer unlinks the node, readers should only have an opportunity to observe the data structure in one of three ordered consistent states: without the node unlinked, with the node unlinked but not yet reclaimed, and finally without the node entirely. If a reader can observe a state with the node reclaimed but not unlinked, then causal ordering is violated. To prevent readers from observing this inconsistent state, a writer can wait for a grace period to complete after unlinking the node (and before reclaiming it). When a writer waits for a grace period to complete, it guarantees that readers will not observe any states introduced by writes *after* the grace period before they observe any states introduced by writes *before* the grace period. That is, the pre-grace-period states happen before [16] the post-grace-period states.

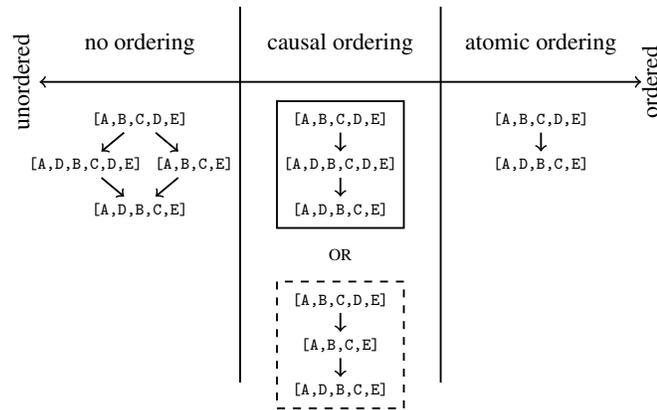
Unlinking a node and then reclaiming its memory is a common example of a sequence of causally related writes, but it is by no means the only example. Relativistic red-black trees [11] and hash tables [30] both rely on grace periods for causal ordering beyond memory reclamation. One simple example of a sequence of causally ordered updates is moving a node to a different position in a singly linked list by inserting a copy at the new position, then unlinking the node at its old position. If we choose to causally relate these operations, we buy a guarantee that readers will see the list in one of three ordered consistent states: with the node at its original position, with the new copy at the new position *and* the original node at the old position, or with the new copy at the new position and the original node unlinked. By preventing readers from observing the list with no new copy at the new position but the original node unlinked, we gain the guarantee that no reader will observe the node temporarily disappearing during a move operation. In exchange, we lose the

guarantee that no reader will observe two copies of the same node in a traversal, because we allow a state where the node is visible in both its new and old positions.

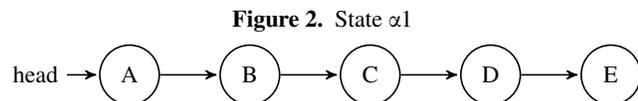
**Example 1** Consider a singly linked list [A,B,C,D,E] that reader threads are constantly traversing *only* from left to right. The readers hold a shared reference to the first node (head), and must start each traversal there. We (the writer) can use a grace period to enforce causal ordering when we move node D from a later position (after C) to an earlier position (after A). Once the move is complete, the readers will observe the modified list [A,D,B,C,E].

Figure 1 shows the partial orders on list states readers can observe during this move operation. If we do nothing to enforce ordering, the left column applies. If we enforce causal ordering, one of the orders in the middle column applies (in this example, we choose the order in the solid box instead of the order in the dashed box, to guarantee that readers don't miss D during the move). If we enforce atomic ordering, for instance using STM, the order on the right applies.

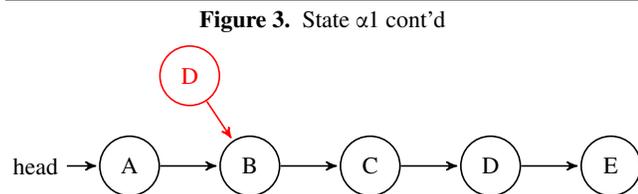
**Figure 1.** Free through atomic orderings for Example 1 – arrows are transitions between observable states



The initial list is shown in Figure 2 (State  $\alpha_1$ ).

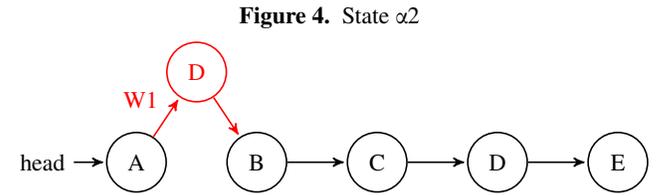


First, we create a new copy of D in freshly allocated memory. We write to memory to do so, but no nodes in the shared list point to it yet, so we know readers have no way of reaching it.



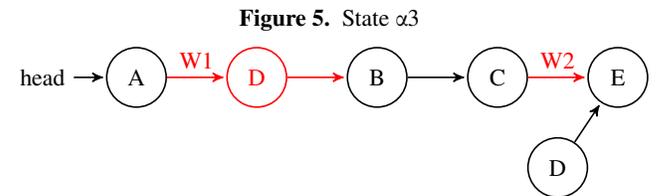
Next, we link the new D into the list by updating the reference in A to point to the new D. This reference is reachable from head, so we have marked its update as **W1** since it is the first of two causally ordered writes that are visible to readers. Now we have put the list in a new state (State  $\alpha_2$ ) where the new copy of D is linked in at its new position, and the old copy remains at its original position. Readers concurrently traversing will either see State  $\alpha_2$  ([A,D,B,C,D,E]),

or have already passed the node we changed, and will see State  $\alpha_1$  ([A,B,C,D,E]).

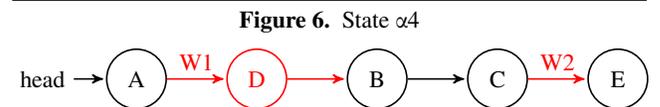


Now we've arrived at an impasse. We need to remove the old copy of D by pointing the reference in C to E (to obtain State  $\alpha_3$ ), but a concurrent reader may have passed A before we pointed its reference to the new copy of D, and then immediately been preempted, performed a long-running computation, or been otherwise delayed in flight. If we update the reference in C to point to E before the leisurely reader visits C, that reader will also miss the old copy of D, and see the list in an inconsistent state ([A,B,C,E]). But we need to unlink the old D to finish our move operation!

We need a guarantee that there are no readers in flight who missed the new D. Another way of saying that is that we need a guarantee that there are no readers who have missed the write that creates State  $\alpha_2$ , but may see the write that creates State  $\alpha_3$ . Yet another way of saying that is that we need to guarantee that State  $\alpha_2$  happens before State  $\alpha_3$ . So we must wait for a grace period to complete, and *then* make a second write **W2** to unlink the old D. This satisfies the guarantee, since every reader that missed the new D has finished its traversal and seen State  $\alpha_1$ , every reader that saw the new D has finished its traversal and seen State  $\alpha_2$ , and any reader who starts a new traversal must do so at head, and will see the new D.



Now the move operation is complete. Readers may still have seen the reference in C before we updated it to point to E, so they may still visit the old D, so we can't reclaim the memory it occupies yet. But conveniently, the GHC runtime garbage collector will automatically reclaim that memory after readers are finished with it.

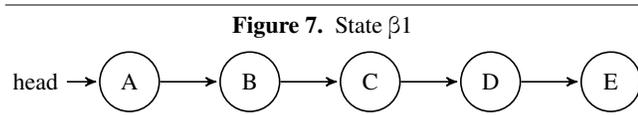


At a high level, we made two causally ordered writes (**W1** and **W2**) going from left to right in the list. Readers are traversing the list left to right, and because our writes also progressed from left to right, we had to wait for a grace period to complete between them to guarantee that no reader could observe **W2** (which created State  $\alpha_3$ ) without observing **W1** (which created State  $\alpha_2$ , and is also part of State  $\alpha_3$ ). The current version of Monadic RP does not statically reject programs that omit a grace period in this situation, as demonstrated in Section 5.3. This paper describes ongoing work to address this at the end of Section 2.1.2. Although we need a grace period to guarantee causal ordering when writes are committed in the same order as reads, when the order of writes is *opposite* the order of reads we do not; Section 2.1.2 explains why.

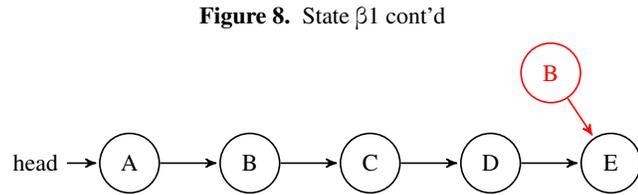
### 2.1.2 Reader traversal order can also enforce causal ordering

Grace periods enforce causal ordering because a writer waits for readers to complete their critical sections before it makes its next causally ordered write. This slows writers down, reducing concurrency between readers and writers, and could create a bottleneck in write-heavy workloads. So it makes sense to avoid grace periods when we have an opportunity to do so. When we can create the states we need using a sequence of writes to nodes in the opposite order of reads, and have guarantees that writes are committed in program order and dependent loads are not reordered, we have such an opportunity.

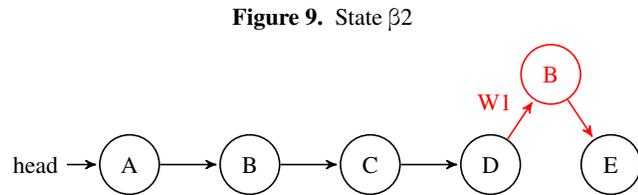
**Example 2** Consider the same singly linked list as before [A,B,C,D,E], with readers traversing again left to right. This time, we will move B from an earlier position (after A) to a later position (after D). After the move operation is complete, readers will observe the modified list [A,C,D,B,E].



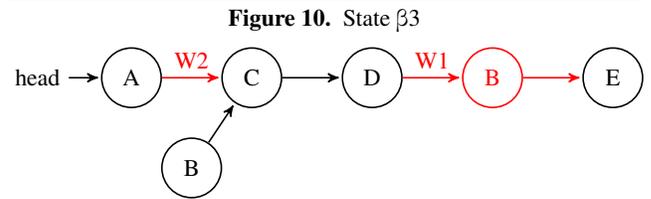
First, we create a new copy of B to link into the list.



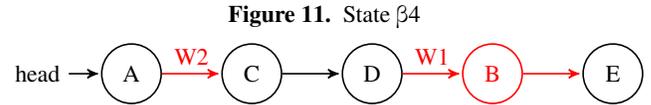
This time,  $W_1$  updates the reference in D to link the new B into the list. Readers concurrently traversing will either observe  $W_1$  and see State  $\beta_2$  ([A,B,C,D,B,E]), or miss  $W_1$  and see State  $\beta_1$  ([A,B,C,D,E]). The key difference from the previous example is that the first write is at the later position in reader traversal order, instead of at the earlier position.



Last time, we had to wait for a grace period to complete after  $W_1$ , because it was possible for a reader to see  $W_2$  but miss  $W_1$  in a traversal, since readers visit the site of  $W_2$  after visiting the site of  $W_1$ , and both writes could occur between those visits. This time,  $W_2$  is at an earlier position than  $W_1$ , so if a reader sees  $W_2$ , it will definitely see  $W_1$  (and observe State  $\beta_3$ ). This is because we committed  $W_1$  before  $W_2$ , the reader has seen  $W_2$ , and has yet to visit the site of  $W_1$ , and so could not have missed  $W_1$  already. If the reader misses  $W_2$ , it may see  $W_1$  (and observe State  $\beta_2$ ), or it may miss  $W_1$  (and observe State  $\beta_1$ ).



As before, the garbage collector reclaims the old D at some point after no thread holds a reference to it.



This time, because our writes progressed in the opposite direction from reads, we did not have to wait for a grace period to complete between writes to guarantee that readers saw State  $\beta_2$  happen before State  $\beta_3$ . This guarantee rests on two conditions:

- Most RP implementations (including Monadic RP) guarantee that writes are committed in program order, so we know that, if  $W_2$  is available to readers, then  $W_1$  is as well.
- Readers visit the site of  $W_2$  before they visit the site of  $W_1$ . So if a reader sees  $W_2$ , that means that  $W_1$  is available, so when the reader subsequently visits  $W_1$ 's site, it is guaranteed to see  $W_1$ .

Note that this does not mean that readers will see the nodes we changed in the order we changed them. Readers only observe nodes in traversal order. In this case, that means that they will observe the nodes we changed in the opposite of the order we changed them. Not exactly intuitively, this is the property that guarantees our most recent write happens before our previous ones.

In ongoing work, we are developing a *domain-specific embedded language (DSEL)* [2] that tracks relativistic reader traversal direction and causally ordered write sequence direction using type-level state in parameterised monads [1] implemented using GHC type families [26], as demonstrated by Kiselyov et al. [14]. This DSEL prevents construction of type-correct programs that make sequences of causally ordered writes in traversal order without invoking `synchronizeRP` between each pair. However, it does allow programs that make sequences of causally ordered writes against traversal order that do not invoke `synchronizeRP` between each pair.

### 2.2 RP Discipline

Researchers have identified a set of rules for correct relativistic programming [21, 29], which we have adapted to match the terms and abstractions this work describes:

1. A writer must only make changes that leave shared data in a consistent state, and guarantee that readers always observe the consistent states that causally related changes create in order.
2. A thread can read from shared data only:
  - (a) inside a write-side critical section, or inside a read-side critical section that announces its end to the world, so grace periods track all readers; and
  - (b) using RP read primitives, to prevent memory access reorderings that may lead to inconsistent observations.
3. A thread can write to shared data only:
  - (a) in a context where writer concurrency can be controlled if necessary, such as a write-side critical section; and

- (b) using RP write primitives, to prevent memory access reorderings that may lead to inconsistent state publications.
- 4. A thread must not wait for a grace period to complete inside a read-side critical section. Such a thread would deadlock (waiting for its own critical section to end) and cause any threads that subsequently wait for a grace period to complete to also block forever [19].
- 5. A thread must not use references obtained during a critical section outside that critical section.

This is not an exhaustive list of rules in the discipline for correct RP. This work provides and proposes static checks for Rules 1–4, but only proposes an incomplete solution for 5.

### 3. Contribution

Most existing work on RP correctness, discussed in Section 6.1, focuses on proving that an RP *implementation* is correct, is a sound foundation. Checking that relativistic *programs* obey the RP discipline is another matter entirely. While researchers have proposed candidate rules for static checking (see Section 6.3) and implemented runtime checks (see Section 6.2), this problem remains unsolved.

There is room in this design space for an implementation that

- can be used to write general-purpose relativistic programs, and
- automatically statically checks that these programs follow the RP discipline.

With these goals in mind, we introduce *Monadic RP*, a Haskell implementation of relativistic programming that uses types to mark and separate read-side and write-side critical sections and to restrict the programmer to the minimal set of effectful operations required in each. Programmers can write new relativistic programs in Haskell using the abstractions Monadic RP provides, which ensure that these programs meet the safety guarantees in Section 3.3.

#### 3.1 Explicit Critical Sections

Many existing RP implementations [4, 7, 27] track grace periods using explicitly marked read-side critical sections, and control writer concurrency using explicitly marked write-side critical sections. Monadic RP adopts this approach, and uses these demarcations to solve three problems:

1. If read-side critical sections write to shared data (a programmer error), they cannot safely proceed concurrently with writers or one-another. Rule 3a prohibits this.
2. If a reader waits for a grace period to complete inside a read-side critical section, it will deadlock. Rule 4 prohibits this.
3. If writers update the same data concurrently, in many cases the data may end up in an inconsistent state. Monadic RP controls writer concurrency by preventing it entirely, which certainly satisfies Rule 3a, but alternative methods (such as STM) would also satisfy this rule, and could increase writer concurrency.

#### 3.2 Expressing RP computations

Monadic RP represents read-side and write-side critical sections with distinct monad types ( $RPR^3$  and  $RPW^4$ ), and represents shared references with type  $SRef$ , which can be read only using `readSRef` and `writeSRef`. It also provides the `synchronizeRP` function, which writers can use to wait for a grace period to complete. These types and functions are defined in the `RP` module.

<sup>3</sup> We color operations in the `RPR` monad `blue`.

<sup>4</sup> We color operations in the `RPW` monad `red`.

The `RP` module exports four monads, each corresponding to a stage in the life cycle of an RP computation. Computations in `RPR` are read-side critical sections, and computations in `RPW` are write-side critical sections. The constructor names for the monads are not exported and labeled “Unsafe”, since they can be used to run arbitrary effectful IO computations.

The `readSRef` and `writeSRef` functions are the only way to access `SRefs`. Because `RPR` and `RPW` are in the `RPRead` typeclass, `readSRef` can be called in each. Because `writeSRef` has a return type in the `RPW` monad (`RPW ()`), it can be called only in `RPW`.

Critical sections must run in a thread. That thread may run multiple critical sections in some order and perform computations outside of them using the values they return; the programmer defines these thread computations in the `RPE`<sup>5</sup> monad, read-side sections are wrapped in `readRP` and write-side sections in `writeRP`. Within a write-side critical section, a writer may need to guarantee that some writes are causally ordered with respect to others. The writer invokes `synchronizeRP` to wait for readers, which is available only in the `RPW` monad.

Finally, a relativistic program must initialize the references its threads share with `newSRef`<sup>6</sup>, spawn those threads with `forkRP`, and wait for them to complete and return values using `joinRP`. The programmer defines these computations in the `RP` monad.

These abstractions are sufficient to implement simple RP algorithms with writers that proceed one at a time while readers proceed concurrently. Other implementations allow concurrency between writers to disjoint data [10]. This code skeleton illustrates the structure described above.

```
do state ← buildInitialState
  reader ← forkRP $ do
    x ← readRP $ readSideCriticalSection state
    return $ pureComputation x
  writer ← forkRP $ do
    writeRP $ writeSideCriticalSection state
  x ← joinRP reader
  joinRP writer
```

The tests in Section 5 are implemented as variants on this structure that spawn multiple reader threads.

We have not yet attempted to determine whether the various monads introduced in this work satisfy the standard monad laws. We leave this as a topic for future work, observing that a formal proof would likely require a semantics that can account for concurrency and the inherent nondeterminism of relativistic reads.

Also, we have not yet attempted to add support for a mechanism relativistic threads could use to communicate with threads computing in IO, at least outside critical sections. This or some other limited IO support is necessary to realize many use cases well-suited to RP, such as implementing a DNS server [33].

Appendix A.1 provides the RP source code in full.

#### 3.3 Rules enforced

Monadic RP exports types and functions that provide the following guarantees.

- The only way to create or access `SRefs` is through the `RP` module’s interface. Also, `SRefs` cannot be accessed outside the relativistic computation that creates them. This is necessary to enforce Rule 1, since threads that aren’t part of a relativistic computation could corrupt shared data or observe it in an inconsistent state otherwise.

<sup>5</sup> We color operations in the `RPE` monad `plum`.

<sup>6</sup> We color operations in the `RP` monad `burnt orange`.

- `readSRef` can be run only in `RPR` or `RPW`, and is the only way to read from shared data. These properties enforce Rule 2.
- `writeSRef` can be run only in `RPW`, and is the only way to write to shared data. These properties enforce Rule 3.
- `synchronizeRP` can be run only in `RPW`. This enforces rule 4.
- Arbitrary effectful computations in IO cannot be run in any of the monads that the RP module exports. This is necessary to enforce any of the rules, since a computation in IO can break them all.

For instance, the `snapshot` function in the `RPR` monad implements a read-side critical section that traverses a relativistic linked list, returning a snapshot of the current list state. We used this function to implement the tests in Section 5.

```
data RPList s a = Nil
                | Cons a (SRef s (RPList s a))

snapshot :: RPList s a → RPR s [a]
snapshot Nil = return []
snapshot (Cons x rn) = do
  l ← readSRef rn
  rest ← snapshot l
  return (x : rest)
```

The `snapshot` function type-checks, since it only calls effectful functions in `RPR`.

If we add a `writeSRef` call to this read-side critical section, we cause a type error, since `writeSRef` is only in `RPW`, but `snapshot` is in `RPR`.

```
snapshot :: RPList s a → RPR s [a]
snapshot Nil = return []
snapshot (Cons x rn) = do
  l ← readSRef rn
  writeSRef rn (Cons undefined rn)
  rest ← snapshot l
  return (x : rest)
```

Error:

```
Couldn't match expected type    RPR s a0
with actual type                RPW s ()
```

The same is true for `synchronizeRP`, and all other effectful functions not in `RPR`.

The `prependA` function inserts an 'A' at the beginning of a relativistic string.

```
prependA :: SRef (RPList Char) → RPW ()
prependA head = do head' ← copySRef head
                  writeSRef head (Cons 'A' head')
```

Because `prependA` has a return type in the `RPW` monad, it can only be run inside a write-side critical section.

As Figure 12 shows, the operations available in monads besides `RPW` are severely restricted.

**Figure 12.** A summary of the operations allowed in each monad.

	RP	RPE	RPR	RPW
<code>newSRef</code>	•			•
<code>readSRef</code>			•	•
<code>writeSRef</code>				•
<code>copySRef</code>				•
<code>synchronizeRP</code>				•
<code>forkRP</code>	•			
<code>joinRP</code>	•			
<code>readRP</code>		•		
<code>writeRP</code>		•		

### 3.4 Comparisons to other implementations

Monadic RP statically prevents rule violations that other implementations do not. For instance, the `urcu` user-level RCU library in C of Desnoyers et al. [4] allows any function to be called in any block of code. We illustrate this fact in a `urcu` program.

The `urcu` library includes a test suite with a test for the version of `urcu` most similar to Monadic RP, called `test_qsbr.c`. This test spawns a configurable number of readers and writers who share a single reference. Each writer repeatedly allocates a new block of memory, stores a sentinel value in it, swaps the address of the new block with the shared reference, waits for a grace period to complete so no reader still holds a reference to the old block, and then stores a second sentinel value in the old block. Each reader polls the shared reference, failing if it ever observes a block with the second sentinel. The test runs for a configurable length of time. If any reader fails, it is evidence that the grace period mechanism or memory access primitives do not preserve causal ordering and consistent states. We have modified this test program to show that `urcu` does not statically enforce RP rules Monadic RP does:

- The modified program reads from a shared reference outside a critical section using `rcu_dereference()`, the analogue of `readSRef`. This violates Rule 2a, and Monadic RP prevents this error because `readSRef` can run only in `RPW` or `RPR`, both of which represent critical sections.
- The modified program writes to a shared reference inside a read-side critical section using `rcu_assign_pointer()`, the analogue of `writeSRef`. This violates Rule 3a, and Monadic RP prevents this error because `writeSRef` cannot run in `RPR`.
- The modified program waits for a grace period to complete inside a read-side critical section using `synchronize_rcu()`, the analogue of `synchronizeRP`. This violates Rule 4, and Monadic RP prevents this error because `synchronizeRP` cannot run in `RPR`.

This modified program compiles without warnings or errors, and fails at runtime. C does not include language-level mechanisms to restrict the statements that appear in blocks of code beyond ensuring that they are syntactically well-formed and checking that parameter and return types match argument and result types, respectively, so this result is unsurprising, and does not represent an oversight on the part of Desnoyers et al.. Appendix A.3 contains the text of this modified version of `test_qsbr.c`.

Howard and Walpole [10] used STM to manage concurrent relativistic writers, so we are investigating using GHC STM [8] for this purpose. Although GHC STM is strongly atomic, and so in normal use prevents concurrent reads and writes to the same data, GHC provides the `readTVarIO#` primitive, which allows threads to read transactional variables outside of transactions.

### 3.5 Tracking traversal and update direction

The reader (a person, not a thread) may realize that the abstractions Monadic RP provides for reading and updating shared data do not capture any information about traversal or update direction. Because of this, there is no way to track whether `synchronizeRP` or updates against traversal order are correctly used to guarantee causal ordering, as illustrated in Section 2.1, and enforce Rule 1. In ongoing work to address this shortcoming, we are developing a cursor-based interface for critical sections, where the cursor starts at a shared reference at the beginning of each critical section, and threads must explicitly move the cursor to a node to access it. Inside write-side critical sections, this interface forces the programmer to explicitly mark causally ordered sequences of writes. This interface, combined with the type-level representation of traversal and update direction described at the end of Section 2.1.2, will statically enforce more of

the rules from the relativistic programming discipline on programs built using Monadic RP. Specifically, it:

- rules out causally ordered updates in traversal order that aren't separated by `synchronizeRP`,
- allows causally ordered updates against traversal order that aren't separated by `synchronizeRP`,
- prevents `synchronizeRP` from being invoked outside sequences of causally related writes, and
- rules out other programmer errors, such as invoking `synchronizeRP` twice with no intervening writes or invoking `synchronizeRP` before making the first in a sequence of causally ordered writes.

We believe that this approach is applicable to relativistic trees because we can generalize the notion of traversal direction to distinguish between expanding the frontier of explored nodes in keeping with a partial order where parents precede children, and backtracking to nodes already within this frontier. This distinction fits naturally with the zipper abstraction.

## 4. Implementation

The RP module exports functions built on top of IO, but does not allow arbitrary IO computations inside relativistic computations. The types we define in the RP module, and the choice of which names to export from it, enforce these controlled abstractions. Additionally, the primitives we use to implement Monadic RP meet the memory ordering requirements of Rules 2b and 3b.

### 4.1 Safe effectful abstractions in Haskell

The IO monad provides all general effectful operations, including file I/O, direct manipulation of memory, and thread management. Because relativistic programs directly mutate shared memory and spawn threads, Monadic RP must be implemented using functions in IO. However, if all computations in IO are allowed inside `RPR` or `RPW`, programmers could violate the RP discipline. For instance, readers could write to arbitrary memory and wait for grace periods to complete.

Terei et al. [28] introduce Safe Haskell, and demonstrate a flexible technique for defining a restricted abstraction RIO on top of IO where clients of RIO can access IO operations only through a fixed set of predefined functions exported by RIO. The RP module is implemented using this abstraction-building technique.

### 4.2 Algorithm

Monadic RP implements grace periods using *quiescent states* [23], in an approach modeled after the *Quiescent-State-Based Reclamation (QSBR) RCU* implementation of Desnoyers et al. [4]. Their work outlines the implementation in detail, so we present a summary here.

In this formulation of Quiescent-State-Based RP, grace periods are announced and completed using a set of counters. There is a global grace period counter, and every thread has its own local counter. We can think of the local counters as a variant of a logical clock [16] where there are only global events, represented in shared memory. The only event that increments any counter is when the thread currently inside a write-side critical section (the writer) increments the global counter before waiting for a grace period to complete. When the writer increments the global counter, it sends a message to readers that a grace period is begun. In response, each reader sends a message to the writer that it has finished a critical section (entering a quiescent state) by updating its local counter to match the global counter. Once the writer has received this message from each active reader by blocking on the reader's local counter until the reader updates it, the writer moves on with its write-side

critical section. Additionally, a thread can opt out of participation in the clock by setting its local counter to zero, allowing grace periods to complete without any message from the thread. Desnoyers et al. [4] say a thread which has opted out is in an extended quiescent state, or offline. A thread always opts out at the beginning of a write-side critical section. This is because it is now the writer, and would not be able to make progress while deadlocked waiting for itself to complete a read-side critical section. At the end of a write-side critical section, the thread (having laid down its writer's mantle) returns its local counter to its previous state.

### 4.3 Primitives

The RP module depends on two memory barrier primitives built into the GHC runtime and exposed to Haskell programs by the `atomic-primops` package. Both act as full compiler-level memory barriers, preventing both GHC and the C compiler that generates the GHC runtime code from reordering reads or writes across either barrier, but these barriers have different semantics at the hardware level.

One barrier primitive is called `writeBarrier`, and prevents hardware reordering of writes. On x86, AMD64, ARM (pre v7), and SPARC, writes from a particular core are not reordered, so GHC [17] correctly implements `writeBarrier` using no instruction at all on these architectures. On PowerPC, and ARMv7 and later, GHC implements `writeBarrier` using the appropriate instructions along with a C compiler reordering barrier.

The `writeSRef` function calls `writeBarrier` before it writes to memory. This guarantees that writes are committed in program order, which in turn means that writes against traversal order guarantee causal ordering, as illustrated in Section 2.1.2. The `synchronizeRP` function uses `writeBarrier` to ensure that a writer commits the write that increments the global counter before the writer begins waiting for readers to observe this write.

The other barrier primitive is called `storeLoadBarrier`, and prevents hardware reordering of reads and writes. GHC implements it using appropriate instructions on all supported architectures. On x86 and AMD64, GHC uses a `lock`-prefixed `add` instruction that adds zero to the word at the top of the thread's stack (a no-op). On Intel architectures, the `lock` prefix prevents hardware reordering of reads and writes across the instruction it modifies [12]. Although legacy Intel architectures lock the entire bus when executing any lock-prefixed instruction, modern Intel architectures do not when the instruction accesses a memory location cached in exclusive mode on the core executing the instruction (such as the top of the thread's stack), so on modern Intel architectures, this primitive does not incur the cost of bus locking.

The `synchronizeRP` function uses `storeLoadBarrier` to insulate the code that waits for a grace period to complete from write-side critical sections. The `readRP` function (which encapsulates a read-side critical section) uses `storeLoadBarrier` to insulate a reader thread's updates to its local counter from the memory accesses inside the current or subsequent critical sections.

### 4.4 Memory Reclamation

Kung and Lehman [15] demonstrate that RP implementations can rely on a general-purpose garbage collector to guarantee that node unlinking and node memory reclamation are causally ordered. GHC has a *tracing garbage collector* [18] that will not reclaim memory some thread could still reach, so Monadic RP does not need to use any additional causal ordering mechanism to prevent premature reclamation.

## 5. Results

We have implemented the examples from Section 2.1 using Monadic RP, and used these implementations to test our causal ordering mechanisms. We tested three hypotheses:

1. our implementation of `synchronizeRP` ensures causal ordering between writes in traversal order,
2. our implementations of `writeSRef` and `readSRef` together ensure causal ordering when writes are against traversal order,
3. in a program that does not use either causal ordering mechanism, readers may observe shared data in an inconsistent state.

As in the examples, all tests start with the shared linked list [A,B,C,D,E]. To test hypothesis 1, we implemented Example 1 from Section 2.1.1. To test hypothesis 2, we implemented Example 2 from Section 2.1.2. To test hypothesis 3, we implemented a variant of Example 1 that does not invoke `synchronizeRP` between W1 and W2.

Each test spawns 29 readers and a writer. Each reader traverses the list 400,000 times in a tight loop, then returns aggregate counts of the consistent and inconsistent states it observed during these traversals. The writer simply moves a list element and returns. We ran these tests on a 30-way QEMU guest on top of 40-way host with Intel®Xeon® E5-2670 v2 @ 2.50GHz CPUs. We ran each test 10,000 times.

### 5.1 Example 1

The first test, `moveDback`, implements Example 1 from Section 2.1.1, moving D from a later position to an earlier position. It invokes `synchronizeRP` in between the `writeSRef` call that links the new copy in at the earlier position and the `writeSRef` call that unlinks the old copy at the later position, since these writes are in reader traversal order.

```
moveDback :: SRef s (RList s a) → RPW s ()
moveDback head = do
  (Cons a ra) ← readSRef head -- [A,B,C,D,E]
  -- duplicate reference to B
  ra' ← copySRef ra
  (Cons b rb) ← readSRef ra
  (Cons c rc) ← readSRef rb
  (Cons d rd) ← readSRef rc
  ne ← readSRef rd
  -- link in a new D after A
  writeSRef ra (Cons d ra') -- [A,D,B,C,D,E]
  -- wait for readers
  synchronizeRP
  -- unlink the old D
  writeSRef rc ne -- [A,D,B,C,E]
```

### 5.2 Example 2

The second test, `moveBforward`, implements Example 2 in Section 2.1.2. It moves B from an earlier position to a later position by inserting a new copy at the later position, then unlinking the old copy at the original position. Because these writes are against traversal order, the writer does not need to invoke `synchronizeRP` to enforce causal ordering.

```
moveBforward :: SRef s (RList s a) → RPW s ()
moveBforward head = do
  (Cons a ra) ← readSRef head -- [A,B,C,D,E]
  bn@(Cons b rb) ← readSRef ra
  (Cons c rc) ← readSRef rb
  (Cons d rd) ← readSRef rc
  -- duplicate the reference to E
  rd' ← copySRef rd
  -- link in a new B after D
```

```
writeSRef rd $ Cons b rd' -- [A,B,C,D,B,E]
-- don't need to synchronizeRP,
-- write order against traversal order
-- unlink the old B
writeSRef ra bn -- [A,C,D,B,E]
```

### 5.3 Example 1, without `synchronizeRP`

The third test, `moveDbackNoSync`, is the same as the first, except that it omits `synchronizeRP`.

### 5.4 Test Results

**Figure 13.** Aggregate results from all test runs, including counts of inconsistent states readers observed.

	Example 1 with <code>synchronizeRP</code>	Example 1 w/o <code>synchronizeRP</code>	Example 2
time (hours)	13.93	13.97	13.95
snapshots	116,000,000,000	116,000,000,000	116,000,000,000
consistent	116,000,000,000	115,999,999,547	116,000,000,000
inconsistent	0	453	0

Figure 13 shows that readers observed no inconsistent states during test runs for Example 1 with `synchronizeRP` or Example 2, giving us confidence that both `synchronizeRP` and writes against traversal order ensure causal ordering in this implementation. When we omitted `synchronizeRP` from Example 1, readers occasionally observed inconsistent states, confirming that we need `synchronizeRP` to guarantee causal ordering in Example 1. To create a meaningful corresponding test for Example 2 without causal ordering, we would need to omit the write barrier in `writeSRef` and run tests on a highly concurrent architecture that reorders stores from a given core.

The first and third tests demonstrate that in Monadic RP, `synchronizeRP` and writes against traversal order provide the causal ordering guarantees we describe in Section 2.1. That is, they are sufficient to enforce Rule 1 if used correctly. The second test demonstrates that without using one of these mechanisms, we cannot guarantee causal ordering in the presence of concurrent reads and writes. That is, they are necessary to enforce Rule 1.

## 6. Related work

### 6.1 Verified implementations

Hart et al. [9] and Desnoyers et al. [4] have written in C and benchmarked RP implementations with a variety of grace-period-based causal ordering mechanisms. Two of these implementations and example algorithms were modeled on a virtual architecture with a weakly-ordered memory model in Promela. Assertions about the model, expressed in Linear Temporal Logic, were verified using the Spin model checker [5]. The Promela model was constructed manually and the verification took hours to run; in exchange, Desnoyers et al. were able to detect errors in the implementation at the level of missing memory barriers and algorithm-level race conditions. This verification gives users high confidence that these implementations are correct, but says nothing about new programs built on top of them.

Gotsman et al. [7] implemented RP in a C-like formal language and verified this implementation in a sequentially consistent memory model using an extension of the logic RGSep [32]. This implementation is not intended to be used to build new general-purpose programs.

Tassarotti et al. [27] implemented RP in Coq, and verified this implementation in a release-acquire memory model [13] using the logic GPS (a logic with ghost state, separation, and per-location

protocols introduced by Turon et al. [31]). In this case, the implementation and the proof are in one system, but each function is an annotated Hoare triple with GPS predicates and permissions. A programmer can build new general-purpose programs on top of this implementation, but she would need skill in GPS and interactive theorem proving to prove that these programs are correct as Tassarotti et al. have for the underlying implementation.

## 6.2 Lockdep

McKenney [20] describes Lockdep-RCU, an extension of the Linux kernel lock validator [3] that does flexible runtime assertion checking in and around Linux RCU API calls. For instance, it is configured to emit a warning when `rcu_dereference()`, the analogue of `readSRef`, is used outside a critical section. A Linux developer can build the kernel with Lockdep enabled to turn on this runtime assertion checking, and then run tests to exercise the code paths with assertions. Lockdep does not provide any guarantee that assertions will be checked unless they are encountered at runtime, so it gives less assurance than a static analysis.

## 6.3 Proposed Coccinelle tests

McKenney [21] proposes using Coccinelle scripts to automatically detect a series of “RCU abuses” in Linux, which include:

- waiting for readers in a read-side critical section (a violation of Rule 4);
- accessing an RCU-protected shared data structure through a private reference obtained during a read-side critical section, rather than a shared reference (a violation of Rule 5).

## 7. Conclusion

Monadic RP is a new GHC Haskell library for relativistic programming that uses types and abstractions to guarantee basic safety properties. We have used it to implement working relativistic programs that demonstrate that it enforces causal ordering in two ways: using grace periods, and using writes against reader traversal order. Existing work on RP focuses on developing performant implementations that do not model or enforce safety properties, runtime checks, or on formal models for that verify correctness of the implementation rather than the correctness of programs that use it. Monadic RP stakes out a novel middle ground: it provides an implementation suitable for general-purpose RP that statically enforces rules from the RP programming discipline. We accomplish this by exporting types and functions from the RP module that statically enforce a separation between read-side and write-side critical sections and restrict the operations available in each. Minimizing the set of operations available in each kind of critical section keeps writes and other effects out of read-side critical sections and keeps effects besides reads, writes, and waiting for grace periods out of relativistic programs, making them safer. In ongoing work, we are extending Monadic RP to expose a cursor-based abstraction for critical sections that uses parameterised monads to track reader traversal direction and writer causal update direction, and statically reject relativistic programs that fail to follow rules that guarantee causal ordering.

## Acknowledgments

We would like to thank Dr. Mark Jones, Dr. Andrew Black, Dr. David Maier, Dr. James Hook, Larry Diehl, and Kendall Stewart of Portland State University, for teaching the classes and engaging in the long series of conversations that led to this work. This work is funded by NSF Award No. CNS-1422979.

## References

- [1] R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.
- [2] E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102(2):145–176, 2010.
- [3] J. Corbet. The kernel lock validator. Available: <http://lwn.net/Articles/185666/> [Viewed May 20, 2015], May 2006.
- [4] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel Distributed Systems (USA)*, 23(2):375–82, 2012.
- [5] M. Desnoyers, P. E. McKenney, and M. R. Dagenais. Multi-core systems modeling for formal verification of parallel algorithms. *ACM SIGOPS Operating Systems Review*, 47(2):51–65, 2013. URL <http://dl.acm.org/citation.cfm?id=2506174>.
- [6] A. Einstein. *Relativity: The special and general theory*. Penguin, 1920.
- [7] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, pages 249–269. Springer-Verlag, Springer, 2013.
- [8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM, 2005.
- [9] T. E. Hart, P. E. McKenney, and A. D. Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.
- [10] P. W. Howard and J. Walpole. A relativistic enhancement to software transactional memory. In *Proceedings of the third USENIX conference on Hot Topics in Parallelism (Berkeley, CA, USA, 2011), HotPar*, volume 11, pages 1–6, 2011.
- [11] P. W. Howard and J. Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 26(16):2684–2712, 2014.
- [12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. January 2015. URL <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf>.
- [13] ISO. Jtc1/sc22/wg14. iso/iec 9899: 2011. *Information technology – Programming languages – C*, 2011. URL [http://www.iso.org/iso/iso\\\_catalogue/catalogue\\\_tc/catalogue\\\_detail.htm](http://www.iso.org/iso/iso\_catalogue/catalogue\_tc/catalogue\_detail.htm).
- [14] O. Kiselyov, S. P. Jones, and C.-c. Shan. Fun with type functions. In *Reflections on the Work of CAR Hoare*, pages 301–331. Springer, 2010.
- [15] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions in Database Systems*, 5(3):354–382, Sept. 1980.
- [16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–65, July 1978. ISSN 0001-0782. .
- [17] S. Marlow. Smp.h (ghc source), December 2014. URL <https://github.com/ghc/ghc/blob/master/includes/stg/SMP.h>.
- [18] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, pages 11–20. ACM, 2008.
- [19] P. McKenney. Read-copy update (rcu) validation and verification for linux (slides), November 2014. URL <http://www.rdrop.com/~paulmck/RCU/RCUVal.2014.11.11a.Galois.pdf>.
- [20] P. E. McKenney. Lockdep-RCU. Available: <https://lwn.net/Articles/371986/> [Viewed May 20, 2015], February 2010.
- [21] P. E. McKenney. Opwintro-rcu. Available: <http://kernelnewbies.org/OPWIntro-RCU> [Viewed May 20, 2015], October 2014.

- [22] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.
- [23] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001. URL [http://www.rdrop.com/users/paulmck/RCU/rclock\\\_OLS.2001.05.01c.pdf](http://www.rdrop.com/users/paulmck/RCU/rclock\_OLS.2001.05.01c.pdf).
- [24] P. E. McKenney, S. Boyd-Wickizer, and J. Walpole. RCU usage in the linux kernel: One decade later. Technical report paulmck.2013.02.24, September 2013. URL <http://www2.rdrop.com/~paulmck/techreports/RCUUsage.2013.02.24a.pdf>.
- [25] J. Morris. Recent developments in SELinux kernel performance. Available: [http://www.livejournal.com/users/james\\\_morris/2153.html](http://www.livejournal.com/users/james\_morris/2153.html) [Viewed May 20, 2015], December 2004.
- [26] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *ACM Sigplan Notices*, 43(9):51–62, 2008.
- [27] J. Tassarotti, D. Dreyer, and V. Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *ACM Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 2015. To appear.
- [28] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. Safe haskell. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 137–148, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6.
- [29] J. Triplett, P. W. Howard, P. E. McKenney, and J. Walpole. Scalable correct memory ordering via relativistic programming. 2011. URL [http://pdxscholar.library.pdx.edu/compsci\\\_fac/12/](http://pdxscholar.library.pdx.edu/compsci\_fac/12/).
- [30] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference*, page 11, 2011.
- [31] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 691–707. ACM, 2014.
- [32] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007—Concurrency Theory*, pages 256–271. Springer, 2007.
- [33] Y. A. Wagh, P. A. Dhangar, T. C. Powar, A. Magar, and P. Mali. Lightweight dns for multipurpose and multifunctional devices. *IJCSNS*, 13(12):71, 2013.

## A. Appendix

### A.1 RP module

Available at <https://github.com/anthezium/MonadicRP/blob/master/src/RP.hs>.

### A.2 RPManyListMoveTest

Available at <https://github.com/anthezium/MonadicRP/blob/master/src/RPManyListMoveTest.hs>.

### A.3 test\_qsbr.c

Available at [https://github.com/anthezium/MonadicRP/blob/master/comparison/test\\_qsbr.c](https://github.com/anthezium/MonadicRP/blob/master/comparison/test_qsbr.c).