Project Summary

Overview. For concurrent programs, programmers often face a mismatch between their assumptions about execution and the memory model of a specific architecture. For example, a programmer may need two instructions to execute in order for the program to be correct, yet most architectures execute out of order. Our goal is to enable programmers to specify such assumptions, prove correctness, and run efficiently on a wide variety of architectures. Ultimately, we want to make concurrent programming easier by enabling programmers to *write once, prove once, and run efficiently anywhere.*

Intellectual merit. Our approach uses *specified orders*. Such orders enable a programmer to specify key assumptions, namely required thread-local execution orders for pairs of instructions. Our preliminary results show that such orders are easier to specify, reason with, and optimize than existing mechanisms such as barriers (assembly language), atomic orderings (C++), and volatiles (Java, Scala). Our nexts step are to design a program logic for programs with specified orders and to machine-check proofs of well-known concurrent algorithms. We will achieve generality by working with a variant of one of the weakest memory models that researchers have formalized, namely Crary and Sullivan's Relaxed Memory Calculus (RMC) from 2015. A proof of correctness based on RMC is also valid for any stronger memory model, including most computer architectures and C/C++. In contrast, existing proofs of concurrent algorithms tend to be less general and tied to a specific memory model; when the memory model changes, those proofs have to change as well. So, specified orders offer a promising approach to concurrent programming, and our program logic will enable tractable reasoning and produce general results. But will our results also apply to Java? Our research hypothesis is that they will, but the answer is elusive because of the complexity of the Java memory model (JMM). We will prove that the JMM is stronger than RMC, and then our results for RMC will also apply to the JMM. We will also devise an algorithm that can synthesize the orders that are needed for correct execution with RMC. Such an algorithm enables a programmer to first write code and then use a tool to the synthesize orders that make the code useful across a spectrum of architectures.

Broader impacts. The need for concurrent programming is growing, especially after the multi-core revolution. Our results on specified orders may help concurrent programmers be more productive and produce software of higher quality. This will be of paramount importance for society's software infrastructure upon which we all rely. Programming and reasoning with specified orders has the potential to benefit programmers across much-used concurrent languages. For example, many mainstream languages, such as C++, Java, and Scala, all support shared-memory concurrency and have semantics with weak memory models. Those languages also include the recent language Rust and even highly dynamic languages such as Ruby. We will develop general techniques that are easily applicable to such languages. I will work with a PhD student on the project and I will teach our results to students in an undergraduate course and a graduate course.

Project Description

1 Motivation

The multi-core revolution has increased the need for shared-memory concurrent programming, algorithms, and languages. In response, programmers target multiple cores to speed up execution, algorithm designers develop concurrent libraries, and language designers make these tasks as easy as possible. Despite much progress, achieving general correctness results remains a challenge, because, in addition to reasoning about the nondeterminism of concurrent programs, a programmer must also reason in the context of a *memory model*.

Intuitively, a memory model is an interface that presents an idealized computer architecture, amenable to reasoning about correctness. A proof of correctness *assumes* a memory model, and then tool support *enforces* that memory model on a real architecture. This separation of concerns, proof from implementation, can simplify correctness proofs and decrease duplication of effort.

For example, the work of [18] assumes that the memory model is Sequential Consistency and proves the correctness of many algorithms. Then a tool made to support Sequential Consistency can enforce correctness by, for example, inserting fences [2]. Similarly, Java algorithms rely on volatile variable annotations that the Java virtual machine enforces using fences [31]. Independently, many algorithms have been proven correct for the C/C++ memory models [50] that are enforced by compilers such as GCC and Clang.

However, the separation of concerns can also negatively impact performance. For example, many of the proofs in Herlihy and Shavit's book remain valid for memory models that are weaker than Sequential Consistency. The reason is that each of those proofs relies on a few assumptions, particular to the algorithm, that are implied by Sequential Consistency. Thus, enforcing Sequential Consistency is too heavy handed and results in insertion of unnecessary fences that hurt performance.

All this leads us to ask: *how can we get both correctness and performance?* Can we have a single memory model yet also leverage algorithm-specific assumptions to maximize performance? In preliminary work [5], published at OOPSLA 2015, we showed example programs for which program-specific assumptions can be stated easily and enforced inexpensively. Those assumptions are execution orders that must be enforced. Our hypothesis is that such orders can enable reasoning about correctness that is of the same complexity as proofs based on Sequential Consistency.

Our next step is to pick a memory model on top of which a programmer can add declarative execution orders. Sequential Consistency is *stronger* than most computer architectures, which is good for reasoning but bad for performance. Instead, we will use a memory model that is *weaker* than most computer architectures, which seems is bad for reasoning but good for performance. We believe that the combination of specified orders and an ultra-weak memory model can lead to the best of both worlds: general reasoning and high performance.

> *Our goal is to enable programmers of concurrent programs to write once, prove once, and run efficiently anywhere.*

Such concurrent programs will have specified orders that express critical assumptions, along with a program logic that enables tractable reasoning, as we explain next.

2 Objectives, Significance, and Broader Impacts

Declarative execution orders. Our notion of specified orders is what we call declarative execution orders. Our slogan is:

> *program = code + declarative execution orders.*

We will illustrate the idea behind declarative execution orders using a variant of the ring buffer that appears in the Linux kernel documentation [21], see Figure 1. The ring buffer implements a queue using a fixed number of memory locations, so when the buffer is full `tryProd` will fail and when it is empty `tryCons` will fail. Figure 1 shows code (`init` plus two columns of lines 1–14), declarative executive orders (six edges that each connects two lines of code), and an illustration of a ring buffer. Each declarative execution order specifies that one instruction must be seen by other threads to execute before another instruction. This will make the code run correctly with the RMC memory model of Crary and Sullivan from 2015. Intuitively, these orders specify what we need for the code to be correct.

Do we actually need to enforce declarative execution orders and if so, how? We will explore this question via examples. If the code in Figure 1 runs with Sequential Consistency, then the behavior required by the six declarative execution orders will be enforced automatically. The reason is that each order is thread-local and goes from an earlier instruction to a later instruction. Thus, Sequential Consistency guarantees the declared execution order because it executes all thread-local instructions in order.

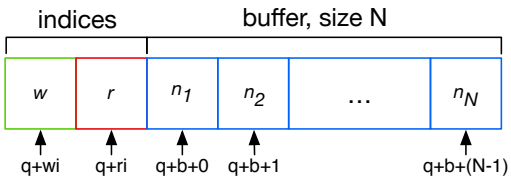
Similarly, if the code in Figure 1 runs on an x86, the six declarative execution orders will happen automatically. The reason is subtle and can be figured out by examining a specification of the x86 memory model, such as the one we give in Figure 3. In contrast, if the code runs on ARMv7 or IA64, a tool must enforce all six execution orders.

We can use a memory fence to enforce an execution order. A memory fence is an instruction that enables us to enforce order of execution for cases when the memory model doesn't already enforce the desired order. Intuitively, the memory operations that appear *before* a fence will execute before the memory operations that appear *after* the fence. For example, on x86 provides the `mfence` instruction, while ARMv7 provides `dmb` and `dmb st`. The choice of a fence and the placement of a fence have impact on performance, as does the *absence* of a fence in case the order is automatically enforced. Recent research [28] goes beyond fences and proposes a hardware-level implementation of a concept akin to declarative execution orders, which turns out to improve performance significantly, compared to memory fences.

In summary, the idea of declarative execution orders is that they can make a program work for a very weak memory model, RMC, and be supported by a fence-insertion tool on stronger memory models. Specifically, in preliminary work we have shown how a tool can take code, orders, and a memory model and insert fences into the code, as needed. We will discuss our approach to fence insertion in more detail in the following section.

The problem. The semantics of a concurrent program depends critically on the memory model. Intuitively, we can think of a semantics as parameterized by a memory model. The weaker the memory model, the more program behaviors are possible, the more efficiently we can execute, and the harder we have to work on correctness. The upshot of a weak

```
init: [q + wi] := 0  [q + b + 0] := 0
      [q + ri] := 0
                     ⋮
                     [q + b + (N-1)] := 0
```




```
1  tryCons(q) =
2   let wic = q + wi in
3   let ric = q + ri in
4   let w = [wic] in
5   let r = [ric] in
6   let limited = w mod N == r mod N in
7   if limited then
8     fail
9   else
10    let index = q + b + (r mod N) in
11    let x = [index] in
12    let _ = [ric] := r + 1 in
13    x
14
```

```
1  tryProd(q, x) =
2    let wic = q + wi in
3    let ric = q + ri in
4    let w = [wic] in
5    let r = [ric] in
6    let w' = w + 1 in
7    let limited = w' mod N == r mod N in
8    if limited then
9      fail
10   else
11     let index = q + b + (w mod N) in
12     let _ = [index] := x in
13     let _ = [wic] := w' in
14     succeed
```

Figure 1: RingBuffer, written in our calculus

memory model is that correctness results carry over to any stronger memory model. If we can overcome the correctness challenges for a very weak memory model, then we take a big step towards *write once, prove once, and run efficiently anywhere.*

Objectives. Figure 2 illustrates our vision. For a program with declarative execution orders, like the RingBuffer in Figure 1, we want to start with the rather extreme case of RMC in pursuit of generality. The code of RingBuffer alone (without the orders) is correct for Sequential Consistency but incorrect for most other memory models. However, once we add the declarative execution orders in Figure 1, the code is correct for the rather extreme case of RMC, hence all the memory models in Figure 2. Then we can travel upward in the lattice of Figure 2 to derive an ideal implementation for each memory model by checking whether all of the orders are necessary. We will pursue the following three research questions.

1. Which logic will support tractable reasoning about programs with a semantics based on RMC and declarative execution orders?

2. Do results about RMC carry over to the Java memory model?

3. Can we design an algorithm for synthesizing a minimal set of declarative execution orders that makes a given program correct?

Thus, our research tasks span logical reasoning, comparison of memory models, and algorithm design. We hope that the often-seen duality of verification and synthesis will be helpful in our setting and lead to new insights. We will apply our logic to prove the correctness of popular concurrent algorithms with declarative execution orders, and, dually, we will use our synthesis algorithm to generate declarative execution orders for such algorithms.

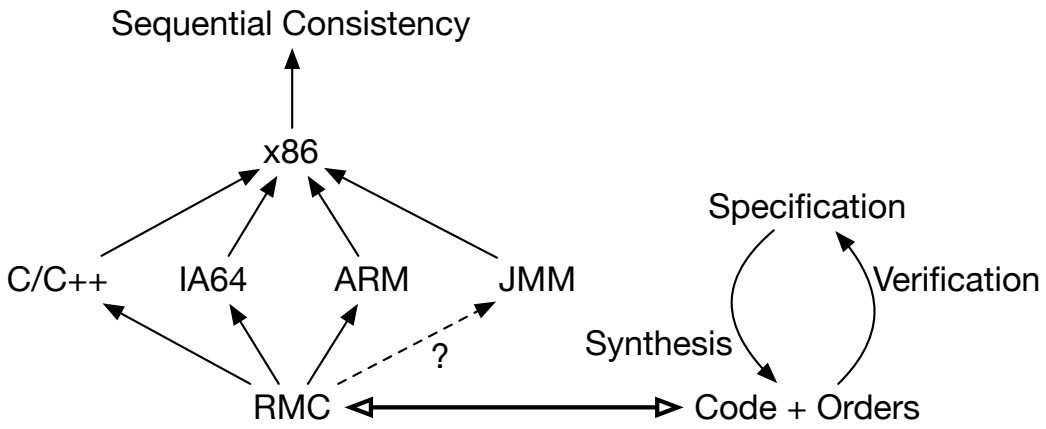


Figure 2: Our vision.

Significance. Our project is significant in three dimensions.

Generality. First, our approach will enable programmers to write their algorithms in a way that make them correct for a wide variety of memory models. Specifically, a program will consist of both code and declarative execution orders that relate directly to the code. Those declarative execution orders enable programmers to think about and state their assumptions in a clear and high-level manner.

Reliability. Second, a programmer can use the rules of our program logic to understand whether a program is correct, without reading the semantics of a memory model. We will support the program logic by an encoding in Coq that enables formal proofs of correctness. Such proofs are done once and for all because of the generality of our approach.

Efficiency. Our use of the RMC memory model means that, from the outset, we enable highly efficient execution. Declarative execution orders can incur a performance cost, but their impact on performance stems directly from correctness considerations. This is better than starting from a blanket assumption of a memory model like Sequential Consistency because we almost always need much less.

Broader impacts. The need for concurrent programming is growing, especially after the multi-core revolution. Our results on specified orders may help concurrent programmers be more productive and produce software of higher quality. This will be of paramount importance for society's software infrastructure upon which we all rely. Programming and reasoning with specified orders has the potential to benefit programmers across much-used concurrent languages. For example, many mainstream languages, such as C++, Java, and Scala, all support shared-memory concurrency and have semantics with weak memory models. Those languages also include the recent language Rust and even highly dynamic languages such as Ruby. We will develop general techniques that are easily applicable to such languages. I will work with a PhD student on the project and I will teach our results to students in an undergraduate course and a graduate course.

3 Preliminary Results

Our preliminary results focus on declarative execution orders. Specifically, we have results on automatic fence insertion (in OOPSLA 2015 [5]) and on the verification of programs based directly on semantics (in recently finished work). We now summarize our results and explain why they represent significant progress.

The history of declarative execution orders. For a long time, researchers have known that correctness can depend critically on the execution order of two instructions. Fences are a crude way of ensuring that two instructions execute in order. They also relate every instruction before them and after them, forcing all the earlier instructions to execute before all the later instructions, which is excessive.

Kuperstein, Vechev, and Yahav [29] used a notion of execution orders as the output of a synthesis algorithm that inspired the synthesis research that we will outline in the next section. Like us, they see execution orders as part of a correct program, but inferred from a correctness property, rather than declared.

The idea of declarative execution orders appeared for first time in publications in 2014–2015, namely in the 2014 PhD dissertation of my student, Mohsen Lesani [32], in the POPL 2015 paper by Crary and Sullivan [12], and in our OOPSLA 2015 paper [5]. Those papers all advocated declarative execution orders as a natural way for a programmer to express intent. Crary and Sullivan's POPL 2015 paper introduced the RMC memory model together with a semantic foundation that includes declarative execution orders.

The results in our OOPSLA 2015 paper. Our OOPSLA 2015 paper asked the question: *How do we enforce the assumptions made by concurrent algorithms?* Additionally, we wanted the enforcement to be parameterized by a memory model, in the style of the ones shown in Figure 3. The task is to enforce the assumptions on the given memory model.

Our solution is an algorithm that first identifies the execution orders that are already enforced automatically by a given memory model, and then inserts fences that enforce the rest. Thus, our algorithm bridges the gap between the specified assumptions and a given weak memory model. We implemented our approach in a tool called Parry that as input takes a concurrent algorithm written in C/C++, declared execution orders, and a memory model, and as output produces C/C++ with fences. Parry passed OOPSLA's artifact evaluation.

Our benchmarks are seven open-source implementations of concurrent algorithms. Three of the benchmarks are transactional memory algorithms for which we first removed all fences and then added specifications of execution orders. Our experiments on x86 and ARMv7 show that Parry inserts fences that are competitive with those inserted by the original authors. For example, for the TL2 transactional memory algorithm [14] on x86, we show that Parry inserts one fewer fence than the experts who implemented the algorithm. Our tool is the first to insert fences into transactional memory algorithms and it solves the long-standing problem of how to easily port such algorithms to a novel memory model. For example, for the Byteeager implementation of the RSTM transactional memory algorithm, the implementers had in mind particular memory models that excluded ARMv7. Parry discovered the need for a fence that would have been missed entirely if one tried simply to modify the fences used for other architectures.

x86	ARMv7	IA64
$\mathsf{st}(x) \mapsto \mathsf{ld}(x)$	$\mathsf{st}(x) \mapsto \mathsf{ld}(x)$	$\mathsf{st}(x) \mapsto \mathsf{ld}(x)$
$\mathsf{st}(x) \mapsto \mathsf{st}(y)$	$\mathsf{st}(x) \mapsto \mathsf{st}(x)$	$\mathsf{st}(x) \mapsto \mathsf{st}(x)$
$\mathsf{ld}(x) \mapsto \mathsf{ld}(y)$	$\mathsf{ld}(x) \mapsto \mathsf{ld}(x)$	$\mathsf{ld}(x) \mapsto \mathsf{ld}(x)$
$\mathsf{ld}(x) \mapsto \mathsf{st}(y)$	$\mathsf{ld}(x) \mapsto \mathsf{st}(x)$	$\mathsf{ld}(x) \mapsto \mathsf{st}(x)$
$* \mapsto \texttt{mfence}$	$* \mapsto \texttt{dmb}$	$* \mapsto \texttt{mfence}$
$\texttt{mfence} \mapsto *$	$\texttt{dmb} \mapsto *$	$\texttt{mfence} \mapsto *$
	$\mathsf{st}(x) \mapsto \texttt{dmb st}$	$\mathsf{st}(x) \mapsto \texttt{sfence}$
	$\texttt{dmb st} \mapsto *$	$\texttt{sfence} \mapsto \mathsf{st}(x)$
		$\mathsf{ld}(x) \mapsto \texttt{lfence}$
		$\texttt{lfence} \mapsto \mathsf{ld}(x)$

Figure 3: Architecture Definitions

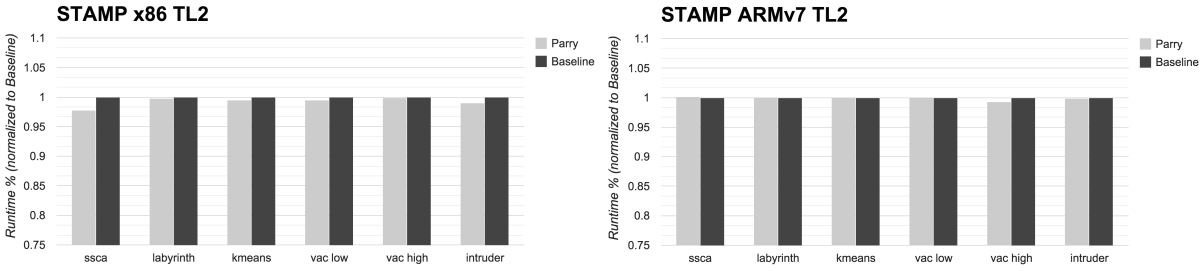


Figure 4: TL2 Performance Benchmarks

Let us consider the performance of the TL2 algorithm, see Figure 4. In the case of x86 we saw a small improvement since we were able to eliminate a fence in `TxCommit`. The improvement is small because `TxCommit` is called infrequently when compared with `TxLoad` and `TxStore` which are the most heavily used procedures in any transactional memory algorithm. Similarly, the results for ARMv7 for the baseline and Parry are nearly identical since the number and placement of fences are nearly identical.

Our approach both assumes and provides modularity. The correctness theorems for concurrent algorithms typically assume *noninterference*, that is, other threads won't access the algorithm's shared state, which therefore is encapsulated. In return, those correctness theorems guarantee that the algorithm will work correctly in any noninterfering context. Thus, our tool inserts fences once-and-for-all that will work in any noninterfering context.

In summary, we found that declarative execution orders are amenable to tool support and lead to satisfying experimental results. In particular, the experimental results confirm that the declarative execution orders can express the intentions of programmers. Additionally, programmers can specify such intention once and for all and then let tool support enforce the assumptions on a given architecture.

Our recent results on verifying programs. We pursue an approach to correctness that is inspired by a 2015 paper [12] on the RMC calculus, which is significantly weaker than other standard memory models. In particular RMC assumes less about the order of write operations than other memory models and it defaults to the semantics of fully relaxed "atomic" memory accesses in the sense of C/C++.

Researchers have used different terminology to describe assumptions about the order of write operations. We use the terminology of RMC and say that the order on writes is a *coherence order*. RMC provides a per-location strict partial coherence order (co) on writes, where nearly all other memory models provide a total order. RMC define the coherence order (co) in terms of four other orders: the program order (po), the reads-from order (rf) that captures information about reads, the visibility order (vis) that captures algorithm-specific assumptions, and finally the happens-before order (hb).

The difference between a total and partial coherence order is significant. A total coherence order enables a semantics to have a notion of *state*. The reason is that after each write operation, we have that all other write operations to the same location either finished earlier or will begin later. In contrast, when working with a partial coherence order, multiple writes to the same location may be under way at any given time which invalidates the entire notion of state and makes reasoning difficult.

The benefit of working in the context of a partial coherence order lies in its generality: a proof of correctness based on RMC carries over to any stronger memory model. We are particularly interested in whether such stronger memory models include the Java Memory Model, which has a partial, rather than a total, coherence order.

RMC uses fully relaxed atomic memory accesses. This makes reasoning about the relative state of multiple memory locations more difficult. Specifically, knowing something about the state of one memory location doesn't provide information about the state of another in the way it would under a stronger memory model such as Sequential Consistency. Sequential Consistency provides the guarantee that all operations will be seen to take place in program order and that all threads see the same sequence of memory operations across locations. That means, when a read executes and associates with some write it gains a wealth of knowledge about the execution in which it appears and the state of many memory locations.

Previous work has carried out proofs under the semantics of release-acquire atomics. The semantics of release-acquire atomics creates *implicit* relationships between accesses to any location before a write with any access after the corresponding read. As a result, proofs in that setting can leverage information learned about one memory location to establish bounds on the state of other memory locations.

With fully relaxed atomics there are no relationships between memory accesses to different locations. This might otherwise make reasoning difficult or impossible but this is a case where declarative execution orders come to the rescue. A declarative execution order is a lightweight way to specify an algorithm-specific relationship between critical memory accesses that is required for correctness.

We write the algorithms in a calculus that we have recently designed; in particular, the ring buffer in Figure 1 is written in our calculus. Figure 5 shows the syntax (which is akin to that of GPS from [51]) and four key rules from the semantics of our calculus (which is like that of RMC from [12]). We use n to range over natural numbers, we use i to range over action identifiers, we use p to range over thread identifiers, we use l to range over labels,

$$
\begin{array}{rrl}
\text{Simple} & s := & v \mid x \\
\text{Actions} & a := & [\,s\,] \mid [\,s\,] := s \mid \texttt{ATM}(s, s) \\
\text{Exp} & e := & a \mid s \mid s + s \mid s \bmod v \mid s == s \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{if}\ s\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \texttt{repeat}\ e\ \texttt{end} \mid l{:}e \mid \texttt{order}(l, l)\ \texttt{in}\ e \mid s = v\ \texttt{in}\ e \mid \texttt{CAS}(s, s, s) \mid \texttt{FAI}(s) \\
\text{Program} & P := & \epsilon \mid P; p : \texttt{fork}\ e \\
\text{History} & H := & \epsilon \mid H, h \\
\text{Events} & h := & \mathsf{exec}(i) \mid \mathsf{rf}(i, i) \mid \mathsf{init}(i, p) \mid \mathsf{edge}(l, l) \mid \mathsf{label}(l, i) \mid \mathsf{is}(i, a) \\
\text{Label List} & L := & \epsilon \mid L, l \\
\text{Transaction} & d := & \emptyset \mid L : i = a \mid i \mid i \text{ to } v \mid l \Rightarrow l
\end{array}
$$

$$
\frac{i_1 \xrightarrow{\mathsf{hb}}_H i_2 \quad \begin{array}{l}\mathsf{writes}(H, i_1, l, v_1)\\ \mathsf{writes}(H, i_2, l, v_2)\end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-hb}
\qquad
\frac{i_1 \xrightarrow{\mathsf{hb}}_H i_r \quad i_r \xrightarrow{\mathsf{po}}_H i_2 \quad \begin{array}{l}\mathsf{writes}(H, i_1, l, v_1)\\ \mathsf{reads}(H, i_r, l)\\ \mathsf{writes}(H, i_2, l, v_2)\end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-hb-thru}
$$

$$
\frac{\begin{array}{ll} & i_1 \neq i_2 \\ i_1 \xrightarrow{\mathsf{hb}}_H i_r & \mathsf{writes}(H, i_1, l, v) \\ i_2 \xrightarrow{\mathsf{rf}}_H i_r & \mathsf{reads}(H, i_r, l)\end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-read}
\qquad
\frac{\begin{array}{ll} i_1 \xrightarrow{\mathsf{rf}}_H i_r & i_1 \neq i_2 \\ i_1 \xrightarrow{\mathsf{co}}_H i_2 & H(\mathsf{is}(i_r, \texttt{ATM}(n_l, n)))\end{array}}{i_r \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-atm}
$$

Figure 5: Syntax and key semantics rules of our calculus

and a value v is a natural number. The coherence order is defined using the four rules in Figure 5. Specifically, in Rule (co-hb), two writes that are to the same location and that are ordered by hb, are also ordered by co. Thus, if a later write has seen the earlier one, they are coherence ordered accordingly. In Rule (co-hb-thru), if a read to some location has seen some write to the same location, hb, and there is a second write program order after the read, po, then the writes are coherence ordered. In Rule (co-read), if an action reads from some write, rf, and if it has also seen, hb after, a second write to the same location, then the second write is before the first write. Intuitively, a read must take the value of the most recent write it has seen. Finally, in Rule (co-atm), if an atomic read-write reads from some write and there is a second write which is coherence ordered after the first write, then the second write is also coherence order after the atomic read-write.

Our recent results began with the challenge: *Given a specification, an algorithm, declarative execution orders, and a memory model with partial coherence orders and fully relaxed atomics, prove that the algorithm is correct.*

In response, we have proved the correctness of the RingBuffer algorithm in Figure 1 and Peterson's algorithm. The critical task in our proofs is that of *write elimination*. The process of eliminating writes proceeds by derivation of *cycles in the coherence order*, which are forbidden by the semantics, hence lead to proofs by contradiction. We use this repeatedly in our proofs of RingBuffer and Peterson. Our proofs are long, tedious, and fill 156 pages. Our goal is to write much shorter proofs with a program logic.

4 Proposed Research

Our proposed research has three facets that complement and support each other. We will design a program logic for reasoning about RMC and declarative execution orders, we will work out the relationship between RMC and the Java Memory Model, and we will devise an algorithm for synthesis of declarative execution orders.

4.1 Program Logic

With our preliminary results we have demonstrated that it is possible to perform proofs for real world algorithms in the context of RMC. To build on this success we wish to raise the level of reasoning in our proofs, which are rather close to the semantics, by constructing a program logic. We hope this will simplify reasoning and highlight the core concepts of our proof method.

There is a substantial body of work from which to draw inspiration. There has been significant progress in concurrent program verification, with a variety of memory models, using Hoare logics and, in particular, Hoare logics equipped with different variants of Separation Logic [7, 15, 16, 17, 20, 19, 24, 27, 34, 39, 40, 52]. Notable, amongst them is the work of Turon et al. [51] which verified several complex algorithms including the Linux kernel's read-copy-update algorithm [50] for the release-acquire fragment of C11. Critically, all of these works target a single memory model. Taking a slightly different approach, though still using an invariant based style, Alglave and Cousot created a logic which can be parameterized by a memory model but requires a new proof for each memory model [1].

Our approach begins with a simple premise drawn from the central question that memory models aim to answer: *which writes can satisfy a read?* We will design a logic to answer that question and use it to prove properties about system state. This stands in contrast to existing invariant based approaches. Instead of starting with the goal of thread-local modularity and adding mechanisms to allow for reasoning with weak memory, we start with the necessary mechanisms to reason about weak memory and aim to build in modularity.

Assertions and rules. Figure 6 describes a strawman for our assertion language and strawman examples of logical rules. The first-order fragment is standard with quantification over natural numbers and label sequences. We denote the location of expressions within a program with $e \ @ \ L$. We denote the equality of expressions in the execution of the program using equalities over labeled sequences and natural numbers with $L = L$ and $L = n$ respectively. Finally, we denote the memory model relations over labeled expressions with $L \xrightarrow{\mathsf{R}} L$.

The logic and the proof method are entirely concerned with the boundary between label equalities and memory relations. We use expression level rules like if-alt and let-bind to locate a read of interest, then we use the rules like read and reads-from to move down to the level of the memory model. We use the memory relations on labeled reads and writes to derive contradictions by value or by cycles in the co relation, which are forbidden. Thus we can reason about system state by eliminating the writes that are permitted for a given read.

$$\begin{array}{rrl} \text{Memory Model Relations} & R := & \mathsf{po} \mid \mathsf{vis} \mid \mathsf{rf} \mid \mathsf{vo} \mid \mathsf{co} \\ \text{Assertions} & A := & \mathsf{false} \mid A \Rightarrow A \mid \forall L,\ A \mid \forall n,\ A \mid e\ @\ L \mid \\ & & n = n \mid L = n \mid L = L \mid L < L \mid L \xrightarrow{R} L \end{array}$$

$$\frac{(\texttt{if}\ s\ \texttt{then}\ l_1{:}e_1\ \texttt{else}\ l_2{:}e_2)\ @\ L}{(L = L, l_1)\ \vee\ (L = L, l_2)}\ \text{if-alt} \qquad \frac{(\texttt{let}\ x = l_1{:}e_1\ \texttt{in}\ l_2{:}e_2)\ @\ L}{L = L, l_2}\ \text{let-bind}$$

$$\frac{[\,s\,]\ @\ L \quad L, loc = l \quad L = n}{\exists L_w,\ L_w \xrightarrow{\mathsf{rf}} L_r}\ \text{reads-from}$$

Figure 6: Logical Assertion Language and Example Rules

Scalability. We are taking a "weak memory first" approach to the construction of our logic and proof method. A key challenge with this approach is that there is no ready-made method for using our theorems at higher levels of abstraction. For example, we may wish to use our correctness proof of Peterson's algorithm to help construct a proof of a concurrent algorithm that relies on a locking mechanism.

We see two possible approaches to this problem. The first involves a tiered logic like that of Sieczkowski et al. [49]. The idea is that the theorems proven in the lower level weak memory logic should match an interface ("fictional sequential consistency" in the case of [49]). That interface allows the the higher level logic to incorporate the lower level judgements.

Alternately we may adopt the notion of linearization points from the work of Herlihy and Shavit [18]. This approach lifts the notion of memory relations, like "happens-before", to the level of methods. This is a global reasoning approach to higher level concurrent algorithm proofs which does recover modular reasoning.

The programs we want to verify. In addition to rebuilding our proofs of the RingBuffer algorithm and Peterson's algorithm there are many other algorithms that appear to be within reach of our logic and proof method. We can continue our work on the algorithms in the paper by Turon et al. [51], such as the bounded ticket lock or the spin-lock.

There is also a wealth of proofs and algorithms written using Java in the book *The Art of Multiprocessor Programming* by Herlihy and Shavit [18]. Providing proofs and orders for these algorithms would extend the reach of the book to weak memory models without requiring a significant rewrite or making the content more difficult to learn.

Finally, we can build on my student's experience with proving correctness for software transactional memory algorithms [32] which are arguably among the most complex lock free algorithms.

4.2 Is RMC Weaker than the Java Memory Model?

A distinguishing feature of RMC is that the coherence order is a strict partial order rather than a total order. While most published memory models have total coherence orders, the Java Memory Model (JMM) is different and has a nontotal coherence order. We conjecture that RMC is *weaker* than JMM, which would enable results about RMC to carry over to JMM. In particular, such a result would imply that algorithms written and proved correct for RMC are also correct for JMM. An alternative conjecture is that RMC is weaker than a large fragment of JMM, which in itself would cast light on how JMM compares with other memory models. Ultimately, if researchers can place JMM, RMC, and C/C++ in a formal constellation of memory models, it will aid future language design decisions. We want to settle these conjectures and thereby take a big step towards better declarative fence insertion, logic-based reasoning, and synthesis for JMM.

A 2005 paper [37] by Manson and his colleagues gave a formal specification of the Java memory model. The paper takes an axiomatic approach and defines constraints on the set of permitted Java executions. The paper aims to ensure sequentially consistent executions in the absence of data races (also known as the DRF guarantee). It also aims to prevent "thin-air" reads while permitting compiler optimizations using a justifying execution mechanism.

We are inspired by the work of Alglave and Cousot [1] which leverages the `cat` specification language to enable the definition of arbitrary memory models based on a few key relations over memory accesses. That work also defines a proof method parameterized by the memory model but requires a new proof for each memory model of interest.

Additionally we have successfully related RMC to the release-acquire fragment of the C11 memory model as described by Turon et al. in [51].

We will now sketch two challenges that we must overcome in order to define the relationship between the JMM and RMC.

Thin air reads. The "thin-air" read problem [4] is a long-standing issue for formal memory models, most notably the C/C++ specifications [6] and its many formal variants [25, 51].

As detailed in [53], thin-air reads can be described as "causal cycles", where data dependencies in two threads can be satisfied by writes further down the opposite thread. In Figure 7a, assuming `x` and `y` are shared, the final state of the `p0` and `p1` should never be $\texttt{x} = 1 \wedge \texttt{y} = 1$, but many formal memory models admit these kinds of executions. Unfortunately the naive solution, ruling out causal cycles altogether, also excludes valid executions that can result from compiler optimizations. For example, if we replace `p0` from Figure 7a with `p0'` from Figure 7b a good compiler will optimize the code to `p0''` and both variables can be set to 1 due to interleaving.

Notably the release-acquire C11 semantics of [51] does not suffer from these causal cycles because it is built around the release-acquire fragment of the C11 memory model. This forces one of the reads for the "atomics" of C11 in a causal cycle to be effectively reading "from the future" and so the semantics forbids it. The JMM forbids causal cycles by requiring a justifying execution in a stronger fragment of the semantics. Further work on the JMM and the justifying execution semantics was done by Jagadeesan et al. in [23] and a similar mechanism was used in the context of C++ by Kang et alin [25].

The JMM rules out thin-air reads by justifying the relationship between a particular

```
1 void p0() {
2   if (x == 1){
3     y = 1;
4   }
5 }
```

```
1 void p1() {
2   if (y == 1){
3     x = 1;
4   }
5 }
```

$$O_{\mathsf{Thin}} = \{\mathsf{ld(x)} \rightarrow \mathsf{st(y)}, \mathsf{ld(y)} \rightarrow \mathsf{st(x)}\}$$

(a) Thin-air Read, preventable with O_{Thin}

```
1 void p0'() {
2   if (x == 1){
3     y = 1;
4   } else {
5     y = 1;
6   }
7 }
```

```
1 void p0''() {
2   y = 1;
3 }
```

(b) Permitted optimization

Figure 7: Thin-air reads.

read and write using an alternate "stronger" execution model. If the read was possible in this second model given execution up to a given point in the weaker model then the read is justified. By contrast, the RMC memory model permits thin-air reads through its speculation mechanism. Taking a simple view of the relationship between RMC and the JMM, this is evidence in favor of the fact that RMC admits more executions than the JMM does, but there are several problems.

First, it's not entirely clear that the mechanism described in [37] by which the JMM rules out thin-air reads has the intended effect [3, 22, 48]. Moreover, even defining a thin-air read is difficult and is often done via examples [4, 47]. We must first determine whether the JMM succeeds in ruling out the necessary bad behavior and, if not, we must reconcile that with the thin-air reads permitted by RMC.

Second, our overarching goal is to write once, prove once, run efficiently anywhere. The act of ruling out thin-air reads (which should never happen in practice) may require algorithm-specific assumptions that are unnecessary for a memory model without thin-air reads. The tool support enforcing the extra, spurious assumptions would need to take that into account to avoid extra synchronization. As an example, the orders O_{Thin} in Figure 7a are unnecessary for ruling out the bad state $x = 1 \wedge y = 1$ for any known architecture. However, under the RMC memory model, they are required and would lead to extra synchronization by a tool that enforces those orders.

Proof method. A natural way to relate two semantics is a simulation proof. If RMC can approximate the states of the JMM at every step then we know it supports the same behaviors. This requires a considerable amount of effort and is made awkward by the axiomatic nature of the JMM semantics. We may consider leveraging the semantics defined by Jagadeesan et al. [23] if we choose to pursue a simulation proof. These type of proofs were cited in their motivation yet their semantics wasn't mechanised.

Alternatively we might derive the axiomatic constraints of the RMC memory model under the assumption of the axioms of JMM. Intuitively, if this possible, then we know that the JMM is the more constrained memory model. For this approach, we can leverage the machine checked semantics of Lochbihler et al. [36] or Huisman et al. [22]. This approach is more direct and less effort intensive but requires a mapping between the two memory models. For example, we may map the synchronized-with relation of the JMM to the reads-from relation of RMC. Our work on relating RMC and the acquire-release fragment of C11 in [51] uses such an approach.

4.3 Synthesis of Declarative Execution Orders

Recall our slogan: *program = code + declarative execution orders*. One approach to write such a program is to first write the code and then add declarative execution orders, as needed, to achieve correctness. For this scenario, we want to *automate* the process of adding declarative execution orders. We call this synthesis, from the point of view that the programmer writes part of the program (the code) and synthesizes the rest (the orders).

Previous work. Kuperstein, Vechev, and Yahav [29], presented promising work on synthesis; they infer execution orders from a program, a correctness property, and a memory model. Their approach first runs a whole-program state-space exploration algorithm that produces a logical formula, then solves the formula to get a set of execution orders, and finally uses those orders to insert fences. Their approach to enforce an order (i_1, i_2) is to insert a fence right after i_1 or right before i_2. The whole-program nature of their approach means that while the inserted fences are sound in the given context, they may be unsound in a different context. Still, their approach can give worthwhile feedback to an algorithm designer who tries to specify a set of orders that are sufficient to prove correctness.

Kuperstein et al. [30], Meshman et al. [38], and Dan et al. [13] have presented approaches to a related synthesis problem, allowing degrees of infinite-state programs, but seemingly without execution orders as an intermediate step towards fences.

Liu et al. [35] presented an execution-based approach to inference, in which they run the program on a memory model and then use the traces to infer fences. This technique can likely be recast to infer execution orders instead.

From rules to constraints. In contrast to the previous work, we will use our program logic as the starting point for order synthesis. The program logic has the advantage that it can prove properties that hold in any context that satisfies stated assumptions. Thus, if we specify the context, then synthesis will produce orders accordingly. Intuitively, we specify the synthesis problem as follows.

Problem:	Order synthesis.		
Input:	A	:	assumption about the context,
	P	:	program code (without declarative execution orders),
	C	:	correctness property.
Output:	O	:	execution orders (stated as an assertion) such that $A \wedge O \vdash P : C$
	or the determination that no such O exists.		

The above specification highlights that we would do well to specify the execution orders O in a way that makes O apply to multiple programs P.

Now let us focus on the task of solving the order-synthesis problem. Our approach is inspired by a common approach to *type inference*. Let us first note that the problem of order synthesis is different from type inference:

Order synthesis:	articulate assumptions needed to prove a correctness property.
Type inference:	articulate program invariants.

However, at the level of algorithmics, order synthesis may be able to borrow one of the main techniques for type inference, namely constraint solving. This requires us to take a preparation step that maps order synthesis from a problem that uses logical rules to a problem that uses constraints. We plan to do that in a way that is standard for type systems. For example, we have mapped several type inference problems to equivalent constraint problems [26, 41, 42, 43, 44, 45] and then devised algorithms for solving those constraint problems. For type inference, the mapping from type rules to constraints is intended to satisfy the following kind of property, where P is a program:

$\forall\ P$: P type checks $\iff$ the constraints generated for P are satisfiable.

This property enables type inference to be done by constraint solving. Inspired by the above, we hope to devise a mapping from program logic to constraints that satisfies the following kind of property:

$\forall\ A, P, C$: $\quad (\ \exists\ O\colon\ A \wedge O \vdash P : C\ ) \iff$
the constraints generated from A, P, C are satisfiable.

Intuitively, the solution to the constraints provides us with the orders O.

Constraint solving. Our logic and the generated constraints will lead us to one of the following three constraint-solving scenarios:

1. *The constraints can be solved by an off-the-shelf constraint solver.* In preliminary work [33] on finding bugs in transactional memory algorithms, we successfully used the Z3 SMT solver so Z3 will be the first we will try.

2. *The constraints require a new constraint solver.* While this is inconvenient, we are ready to develop such a constraint solver, in particular because of our success with developing new constraint solvers for a variety of constraint problems derived from type inference [26, 41, 42, 43, 44, 45]. We will also settle the time complexity of the constraint-solving problem.

3. *The constraint problem is undecidable.* This case may be the most likely and we will try to prove that the problem is undecidable. Additionally, we will devise a search algorithm that can find solutions for the programs and contexts that we care about. We are optimistic about solving this search problem because of our experience with declarative execution orders for well-known algorithm. We have found that for a page of code, the number of needed orders tends to be at most six.

Minimality. When we devise declarative execution orders by hand, we tend to go for the smallest number of them. Can our constraint-solving approach find a smallest number of orders, too? Clearly, for every problem with given A, P, C, if the problem is solvable, then a solution with a minimal-size O exists. We will try to find such minimal solutions and we will also explore that two different solutions may lead to different performance overhead. We will try to devise heuristics that give preference to a lower-overhead O.

5 Curriculum Development Activities

We will integrate results from the project into two existing courses at UCLA.

CS 132 Compiler Construction is an undergraduate course in which each student writes a compiler from a subset of Java, called MiniJava, to MIPS assembly code. I would like to add a two-hour lecture about compilers for concurrent languages. This is entirely feasible with the current lecture schedule: after the last lecture about code generation, the students continue with their project for 3.5 weeks during which I give lectures about advanced topics. I can easily replace one of the current advanced topics with concurrency. The goal of the lecture is get the students to think about the problems faced by a retargetable compiler for architectures with multiple cores. I will cover the idea of concurrent programs with declarative execution orders, the program logic to reason about the correctness of such orders, and the algorithm for fence insertion that is at the core of the compiler. I will write detailed lecture notes and make them publicly available.

CS 232 Static Program Analysis is a graduate course in which I present foundations and applications of static program analysis. The theme of the course is constraint-based analysis. I would like to add a two-hour lecture about static analysis and synthesis for concurrent languages. I will cover the idea of order synthesis, along with our approach to generating and solving constraints. The lecture will be based on the papers that we will publish on the topic.

6 Results from related Prior NSF Support

Jens Palsberg is the PI for "SHF: Small: Typed Self-Application" (Award number 1219240, $493,612, 9/12–2/18).

Intellectual merit. This award led to papers in four consecutive POPL symposia [8, 9, 10, 11]. Each of those papers came with substantial software that is available from our webpages and that each passed POPL's artifact evaluation. The URLs of those webpages are listed in the reference list [8, 9, 10, 11]. Our POPL 2015 paper presented the first typed self-representation in a polymorphic lambda-calculus with decidable type checking. This solved a problem of a kind that Frank Pfenning and Peter Lee in 1991 had concluded "seems to be impossible" [46]. Our 's POPL 2016 paper presented a self-interpreter for a strongly normalizing lambda-calculus, which according to conventional wisdom is impossible! Our POPL 2017 paper presented the first polymorphically typed self-evaluator in a lambda-calculus with decidable type checking. Our POPL 2018 presented the first Jones-optimal and self-applicable partial evaluator whose type guarantees that it always generates type-correct code.

Broader impacts. The award supported Matt Brown who received a PhD in 2017. The project also had participation of an undergraduate student who studied typed paradoxes in System U. The techniques that we developed may help programmers type check their meta-programs, which will lead to higher programmer productivity and more reliable software.

t

References

[1] Jade Alglave and Patrick Cousot. Ogre and pythia: An invariance proof method for weak consistency models. In *Proceedings of POPL'17, ACM Symposium on Principles of Programming Languages*, 2017.

[2] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.*, 39(2):6:1–6:38, May 2017.

[3] David Aspinall and Jaroslav Ševčík. Formalising Java's data race free guarantee. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'07, pages 22–37, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Mark Batty and Peter Sewell. The thin-air problem. `http://www.cl.cam.ac.uk/~pes20/cpp/notes42.html`, 2014. [Online, accessed Nov 2017].

[5] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In *Proceedings of OOPSLA'15, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Pittsburgh, Pennsylvania, Oct 2015.

[6] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[7] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, August 2016.

[8] Matt Brown and Jens Palsberg. Self-representation in Girard's System U. In *Proceedings of POPL'15, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Mumbai, India, Jan 2015. Software artifact at `http://compilers.cs.ucla.edu/popl15/`.

[9] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: a self-interpreter for F-omega. In *Proceedings of POPL'16, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, Florida, Jan 2016. Software artifact at `http://compilers.cs.ucla.edu/popl16/`.

[10] Matt Brown and Jens Palsberg. Typed self-evaluation via intensional type functions. In *Proceedings of POPL'17, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Paris, Jan 2017. Software artifact at `http://compilers.cs.ucla.edu/popl17/`.

[11] Matt Brown and Jens Palsberg. Jones-optimal partial evaluation by specialization-safe normalization. In *Proceedings of POPL'18, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Los Angeles, Jan 2018. Software artifact at `http://compilers.cs.ucla.edu/popl18/`.

[12] Karl Crary and Michael Sullivan. A calculus for relaxed memory. In *Proceedings of POPL'15, ACM Symposium on Principles of Programming Languages*, 2015.

[13] Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *SAS*, 2013.

[14] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208. Springer, 2006.

[15] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP'09)*, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with fsl++. In Hongseok Yang, editor, *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017*, pages 448–475. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[17] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, pages 173–188, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[19] C. A. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Foundations of concurrent kleene algebra. In *Proceedings of the 11th International Conference on Relational Methods in Computer Science and 6th International Conference on Applications of Kleene Algebra: Relations and Kleene Algebra in Computer Science*, RelMiCS '09/AKA '09, pages 166–186, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] C. A. R. Hoare. Towards a theory of parallel programming. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming*, pages 231–244. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[21] David Howells and Paul E. McKenney. Circular buffers. `https://www.kernel.org/doc/Documentation/circular-buffers.txt`, 2017. [Online, accessed July 2017].

[22] Marieke Huisman and Gustavo Petri. The Java memory model: a formal explanation. *VAMP*, 7:81–96, 2007.

[23] Radha Jagadeesan, Corin Pitcher, and James Riely. Generative operational semantics for relaxed memory models. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 307–326, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[25] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of POPL'17, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 2017.

[26] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. Preliminary version in Proceedings of FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[27] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In Hongseok Yang, editor, *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017*, pages 696–723. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[28] P. Kumar, P. Gera, H. Kim, and H. Kim. Louvre: Light-weight Ordering Using Versioning for Release Consistency. *ArXiv e-prints*, October 2017.

[29] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010.

[30] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.

[31] Doug Lea. The jsr-133 cookbook for compiler writers. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf`, 2004. [Online, accessed July 2017].

[32] Mohsen Lesani. *On the Correctness of Transactional Memory Algorithms*. PhD thesis, UCLA, 2014.

[33] Mohsen Lesani and Jens Palsberg. Proving non-opacity. In *Proceedings of TRANSACT'13, Workshop on Transactional Computing*, Houston, Texas, March 2013.

[34] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 385–399, New York, NY, USA, 2016. ACM.

[35] Feng Liu, Nayden Nedeve, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

[36] Andreas Lochbihler. Java and the Java memory model – a unified, machine-checked formalisation. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 497–517, Berlin, Heidelberg, 2012. Springer-Verlag.

[37] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[38] Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *Proceedings on Static Analysis Symposium, Lecture Notes in Computer Science Volume 8723*, pages 237–252, 2014.

[39] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[40] Susan Speer Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1975. AAI7612884.

[41] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proceedings of LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[42] Jens Palsberg and Trevor Jim. Type inference with simple selftypes is NP-complete. *Nordic Journal of Computing*, 4(3):259–286, Fall 1997.

[43] Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[44] Jens Palsberg and Tian Zhao. Type inference for record concatenation and subtyping. *Information and Computation*, 189:54–86, 2004. Preliminary version in Proceedings of LICS'02, IEEE Symposium on Logic in Computer Science, pages 125–136, Copenhagen, Denmark, July 2002.

[45] Jens Palsberg, Tian Zhao, and Trevor Jim. Automatic discovery of covariant read-only fields. *ACM Transactions on Programming Languages and Systems*, 27(1):126–162, January 2005. Preliminary version in Informal Proceedings of FOOL'02, Ninth International Workshop on Foundations of Object-Oriented Languages, Portland, Oregon, 2002.

[46] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137–159, 1991.

[47] Bill Pugh. Causality test cases. `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`, 2017. [Online, accessed November 2017].

[48] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, pages 27–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[49] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A separation logic for fictional sequential consistency. In *European Symposium on Programming (ESOP)*, volume 9032, pages 736–761, 04 2015.

[50] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *PLDI*, 2015.

[51] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. Gps: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of OOPSLA'14, Object-Oriented Programming Systems, Languages and Applications*, 2014.

[52] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[53] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for c11 concurrency. In *Proceedings of OOPSLA'13, Object-Oriented Programming Systems, Languages and Applications*, 2013.