AFRL-RI-RS-TR-2017-232

VERIFIED COMPILATION OF CONCURRENT MANAGED LANGUAGES

PURDUE UNIVERSITY

NOVEMBER 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-232 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
JOHN D. MATYJAS
Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
NOV 2017	FINAL TECHNICAL REPORT	JUL 2013 – JUL 2017

4. TITLE AND SUBTITLE

VERIFIED COMPILATION OF CONCURRENT MANAGED LANGUAGES

5a. CONTRACT NUMBER
FA8750-13-2-0242

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Suresh Jagannathan, Jan Vitek

5d. PROJECT NUMBER
HACM

5e. TASK NUMBER
SC

5f. WORK UNIT NUMBER
ML

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Prime
Purdue University
305 North University Street
West Lafayette, IN 47907

Sub
Northeastern University
360 Huntington Ave
Boston MA 02115

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2017-232

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW -2017-5994
Date Cleared: 28 Nov 2017

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of the Havoc project was to explore new proof techniques and methodologies that would enable scalable and modular verification of modern concurrent programming languages like Java or C#. The efforts undertaken during the lifetime of this effort focused on (a) new proof techniques, specifically the use of refinement methods and tactics to simplify reasoning about interferences in proving invariants about concurrent code; (b) incorporating precise notions of memory models, both at the processor and language level, to enable compilation to exploit and be faithful to language definitions and processor features; (c) new designs for compiler intermediate representations that facilitate mechanized proofs and verification; and (d) a realistic case study that combines these ideas to prove the correctness of a state-of-the-art concurrent garbage collector.

15. SUBJECT TERMS

Program verification, compiler design, concurrency, memory models, theorem proving, garbage collection.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STEVEN L. DRAGER
U	U	U	UU	36	19b. TELEPHONE NUMBER (*Include area code*) (315) 330-2735

Contents

List of Figures

1 SUMMARY

This project was centered on the following challenges. Limited **resusability and abstraction** of certified code: The fact that any change to a system must trigger re-certification is untenable. The project considered mechanisms to raise the level of abstraction in certification so that it becomes possible to reason about correctness at a higher-level using rigorously-defined unambiguous semantics for all intermediate representations in the compilation process, rather than at the generated assembly level. **Scaling** validation and **composability**: The size of modern code bases suggests that different levels of modularity in verification are required. We considered architectures and principles that facilitate modular verification at different levels of granularity. **Concurrency**: Even though concurrency is a pervasive part of modern software and hardware systems, it has often been ignored in safety-critical system designs. A major focus of this effort was centered on how best to reason about concurrency as an intrinsic feature by assuming that all activities execute on multi-core hardware with potentially relaxed memory, relying on precise memory model specifications at both the language and architecture level to reason about possible behaviors.

New verification approaches and methodologies lie at the heart of our answer to these challenges. In particular, ensuring the correctness of the translation from source to target effected by a compiler is a critical pre-requiste to building an automatically certified software stack. The existence of such an artifact would dramatically change the safety-critical application landscape, relieving the need for costly manual inspection of source and binary, enabling a richer class of optimizations, leading to more efficient and scalable applications. Specifically, we addressed the challenges enumerated above in the following ways. *Reusability* and *abstraction* is achieved through the use of high-level type-safe language like Java, rather than a low-level one like C, enabling us to reason about correctness in terms of precise source-language invariants. *Scaling* and *composability* was achieved by defining new modular proof techniques to aid the compiler writer in proving the correctness of optimizations, even in the presence of sophisticated managed (concurrent) runtime services like garbage collectors. Important issues related to concurrency were addressed by refining the existing Java memory model to make it more amenable for incorporation within a verified compiler.

2 INTRODUCTION

There were three major activities undertaken during the lifetime of this effort that built on these activities. The first was to explore the construction and compilation methodology of an intermediate representation capable of facilitating correctness proofs on the behavior of concurrently exeucting runtime services. The second was the development of a precise operationally defined memory model that relates the definition of the Java Memory Model (JMM) with weak microprocessor architectures like IBM's Power. The third was the specification and verification of a state-of-the-art concurrent garbage collector as a substantial demonstration of the efficacy of our ideas. All three activities share the overarching goal of developing strong (mechanically checkable) safety guarantees for high-level language implementations built on top of sophisticated runtime services and architectural platforms.

2.1 Proof Methodology

Our first significant result was the development of a new intermediate representation and associated proof methodology to facilitate certified compilation of high-level managed languages like Java or C#. Managed languages provide intrinsic support for concurrency at several levels. Applications can express concurrent computations using threads and synchronization primitives. Additionally, to improve scalability or performance, elements of the language implementation itself may run concurrently with application threads. The interactions between application threads and the diverse components of the language runtime system are regulated by compiler-injected code snippets. Typical examples of injected code include allocation fast paths, read and write barriers, synchronization fences, and initialization checks. These concurrent snippets are sophisticated, often racy, and must operate correctly in an environment subject to program transformations, both local and global. The subtleties involved in dealing with these low-level code fragments within the context of already complex source and target languages justify the effort of adopting a verified compilation strategy. However, verifying the correctness of a compiler for these kinds of languages is a challenging and ambitious goal as it entails reasoning about the inherently parallel behavior of concurrent operations in the source language, as well as the possibly racy, non-atomic, operations introduced by the compiler. Low-level implementations provide a performant variant of high-level specifications that are exploited by the compiler. Reconciling the dichotomy between these two abstraction layers is key to any feasible verification strategy. To do so, we developed new *refinement* predicates that relate the "high" and "low" definitions of concurrent code. Informally, we say that a low-level statement l *refines* a high-level one h if the execution of both l and h starting from the same state leads to the same final state; furthermore, if executing l admits a trace tr of interleaved actions of other threads, then tr must be admissible as a feasible trace under the execution of h. This notion of refinement guarantees the equivalence of high and low-level code. Given a high-level specification h that captures the atomicity properties implicit in l, the refinement predicate helps the compiler writer devise a proof that l *refines* h.

While recent years have seen progress in compiler verification, much of this work has been for sequential languages like C. The basic correctness argument requires proving that any behavior admitted by the compiled program is also admitted by the source. This is typically shown by a *backward* simulation proof between target and source language semantics. Assuming the source program is safe, a backward simulation demonstrates that any observable behavior produced by the target program is a valid observable behavior of the source program as defined by the source language semantics. Demonstrating such a simulation is complicated by the presence of concurrency. Managed languages add further complications because they often compile a single source memory access to multiple low-level memory accesses, as a result of code *injected* by the compiler. For example, Java compilers typically inject write barriers before each field update to support garbage collection. Indeed, implementation of performant write barriers typically use a non-trivial protocol to communicate with the garbage collector thread, and serves to notify that changes are being done in the object graph. Dealing with concurrency is thus quite challenging since it requires proving concurrent invariants of the underlying implementation of the compiler and runtime system, internal data structures, and communication protocols. The details of these protocols are not visible to the high-level source. Consequently, a naïve approach to verification of injected concurrent code

fragments is not scalable using a standard backward simulation argument.

To address this challenge in verified compilation of managed languages, we developed an *atomicity refinement* methodology that coarsens the granularity of injected pieces of code, thereby simplifying the overall verification of the compiler infrastructure. Our approach facilitates the modular expression of such proofs, making a backward simulation argument feasible by establishing the equivalence of fine-grained and coarse-grained representations of concurrency operations, in isolation of the other components in the program. The refinement enables a simulation argument similar to the ones used to demonstrate the correctness of sequential optimizations, and hence allows such arguments to be effectively applied to potentially racy, lock-free, concurrent code. This particular approach is motivated by the premise that establishing that the high-level specification captures the behavior defined by the source program is substantially easier than directly proving the correspondence between low-level target and source.

2.2 Memory Models

Our second major result concerns the verification of memory models within the compiler toolchain. The Java Memory Model is intended to characterize the meaning of concurrent Java programs. Because of the model's complexity, however, its definition cannot be easily transplanted within an optimizing Java compiler, even though an important rationale for its design was to ensure Java compiler optimizations are not unduly hampered because of the language's concurrency features. In response, the *JSR-133 Cookbook for Compiler Writers*, an informal guide to realizing the principles underlying the JMM on different (relaxed-memory) platforms was developed. The goal of the cookbook is to give compiler writers a relatively simple, yet reasonably efficient, set of reordering-based recipes that satisfy JMM constraints.

To aid our certification effort, we embarked on a formalization of the cookbook, providing a semantic basis upon which the relationship between the recipes defined by the cookbook and the guarantees enforced by the JMM can be rigorously established. Notably, one artifact of our investigation is that the rules defined by the cookbook for compiling Java onto the Power microprocessor are *inconsistent* with the requirements of the JMM, a surprising result, and one which justifies our belief in the need for formally provable definitions to reason about sophisticated (and racy) concurrency patterns in Java, and their implementation on modern-day relaxed-memory hardware.

Our formalization enables simulation arguments between an architecture-independent intermediate representation of the kind suggested by the cookbook with machine abstractions for Power and x86. Our results not only provided fixes for cookbook recipes that are inconsistent with the behaviors admitted by the target platform, but also proved the correctness of these repairs, and enabled us to use these verified recipes within our compiler toolchain.

2.3 Runtime System Verification

Concurrent garbage collection algorithms are an emblematic challenge in the area of concurrent program verification. We considered tackling this problem by proposing a mechanized proof

methodology based on the popular Rely-Guarantee (RG) proof technique. We designed a specific compiler intermediate representation (IR) with strong type guarantees, dedicated support for abstract concurrent data structures, parametric specifications of memory model behavior, and high-level iterators on runtime internals. In addition, we defined an RG program logic supporting an incremental proof methodology where annotations and invariants can be progressively enriched.

We have formalized the IR, the proof system, and have proven the soundness of the methodology in the Coq proof assistant. Equipped with this IR, we were able to prove a fully concurrent garbage collector where mutators never have to wait for the collector.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

This effort is concerned with the *verified compilation* of high-level managed languages like Java or C# whose intermediate representations provide support for shared-memory synchronization and automatic memory management. In this environment, the interactions between application threads and the language runtime (*e.g.*, the schedulers, memory managers, etc.) are regulated by compiler-injected code snippets. Example of snippets include allocation fast paths, read and write barriers, synchronization fences and data initialization checks. For performance, the code injected by the compiler is often sophisticated, and racy, but must nonetheless operate correctly in the presence of program transformations, both local and global. This entails reasoning about the inherently parallel behavior of operations in the source language, as well as the operations introduced by the compiler. A naïve approach would entail examination of all possible thread interleavings, an impractical and non-scalable exercise.

To tackle this problem, we developed a general and flexible atomicity refinement technique that increases the granularity of injected snippets of code, hence facilitating simulation proofs between bytecode computations, and their racy, fine-grained implementations. We illustrate our approach by considering the following low-level code snippet that attempts to acquire a lock (akin to a high-level `monitorenter` bytecode instruction of Java):

```
repeat {
  old := cas(Lock, 0, 1);
  current := old;
  while (current != 0) do
     current := Lock;
} until (old == 0)
```

```
atomic{
   assume(Lock == 0);
   Lock := 1
}
```

In the implementation on the left, lock acquisiton requires potentially multiple iterations of a loop that attempt to change the global variable `Lock` from 0 to 1 through a `cas` (`compare-and-set`) instruction. On the other hand, the code on the right is atomic, and only proceeds if the `Lock` variable is 0 (the semantics of `assume` guarantees that). It is obviously easier to match the semantics of `monitorenter` with the code on the right. Our refinement technique can establish that the atomic piece of code on the right simulates the low-level implementation on the left, simplifying verification burden. Moreover, we designed our framework to be cognizant of weak or relaxed

LANGUAGE : $\mathcal{I}$

$$
\begin{array}{rcl}
s \in \mathcal{I} & ::= & \mathsf{skip} \mid d = op(\vec{r}) \mid s; s \mid \mathsf{if}\ c\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{repeat}\ s\ \mathsf{until}\ c \\
& \mid & \mathsf{load}_{\nu}(d, r) \mid \mathsf{store}_{\nu}(d, r) \mid \mathsf{cas}(d, r, o, n) \mid \mathsf{fence} \mid \mathsf{abort} \\
& \mid & \mathsf{atomic}\ s \mid \mathsf{assume}\ c \mid \mathsf{branch}\ s,\ s \mid \mathsf{loop}\ s \\
\nu & ::= & \mathsf{Local} \mid \epsilon
\end{array}
$$

Figure 1: Syntax of $\mathcal{I}$.

memory model behavior such as those defined by the Total Store Ordering (TSO) relaxed memory model of Intel x86 processor [63], thus rendering it applicable in realistic environments.

We illustrate our technique via three simple, yet representative, rules:

$$\frac{s_1 \text{ is dead}}{s_1; s_2 \preccurlyeq s2}\ \textsc{DeadCode} \qquad \frac{d \text{ is known to be local in the context}}{d := l; \texttt{atomic}\{\ s\ \} \preccurlyeq \texttt{atomic}\{\ d := l; s\ \}}\ \textsc{GrowAtomicLocal}$$

$$\frac{}{d := \texttt{cas}(r, v_0, v_1); \texttt{assume}(v_0 = d) \preccurlyeq \texttt{atomic}\{\ d := \texttt{load}\ r; \texttt{assume}(v_0 = d); r := v_1\ \}}\ \textsc{CAS-success}$$

Rule `DeadCode` states that code which affects variables that are not used later can simply be discarded. Rule `GrowAtomicLocal` states that if a certain variable is known to be local in a certain context, then accesses to this variable can be considered as happening atomically with the code that follows. Finally, `CAS-success` establishes that a successful `cas` operation can be treated as an atomic operation. These rules form the core of the proof that the spin-lock example presented before is soundly abstracted as an atomic block.

We have implemented a certified compiler for Java that implements and proves the soundness of our atomicity refinement technique as an extension of the CompcertTSO verified compiler [76]. Because our technique allows the compiler writer to reason compositionally about the atomicity of low-level concurrent code used to implement managed services, it facilitates verified compilation of non-trivial concurrent runtime components. To demonstrate the applicability of our approach, we have also written a concurrent garbage collector based on the algorithm presented in [23]. A particular characteristic of this garbage collector is that it exploits knowledge about TSO (weak memory) behavior by not adding unnecessary fences whose inclusion would otherwise incur substantial performance penalities. We have proven the atomicity of the pieces of code that are necessary to implement this garbage collector (including write barriers and allocation). In the absence of our TSO-aware refinement methodology, significantly more fences would be necessary to make our correctness proof tractable, resulting in diminishing collector performance. Our initial investigation of how garbage collectors interact with client code was further refined and significantly enhanced in later phases of the project, where we substantially augmented the definitions and capabilities of the intermediate representations used by the compiler to facilitate more sophisticated reasoning about garbage collection behavior.

Because our development was initially framed in the context of the Total Store Order relaxed memory model, ensuring compiler correctness became challenging because high-level actions are translated into sequences of non-atomic actions with compiler-injected snippets of racy code; the behavior of this code depends not only on the actions of other threads, but also on out-of-order reorderings performed by the processor. A naïve proof of correctness would require reasoning over all possible thread interleavings. Instead, we developed a refinement-based proof methodology that precisely relates concurrent code expressed at different abstraction levels, cognizant throughout of the relaxed memory semantics of the underlying processor. Our technique allows the compiler writer to reason compositionally about the atomicity of low-level concurrent code used to implement managed services.

While formalizing language behavior in the context of a hardware memory model like TSO is useful and essential to understanding a realitic certified compilaton strategy, it is insufficient in the context of a language like Java because it fails to capture and express executions defined in terms of the Java Memory Model's view of allowable reorderings. The JMM is intended to characterize the meaning of concurrent Java programs. Because of the model's complexity, however, its definition cannot be easily transplanted within an optimizing Java compiler, even though an important rationale for its design was to ensure Java compiler optimizations are not unduly hampered because of the language's concurrency features. In response, the *JSR-133 Cookbook for Compiler Writers* [49], an informal guide to realizing the principles underlying the JMM on different (relaxed-memory) platforms was developed. The goal of the cookbook is to give compiler writers a relatively simple, yet reasonably efficient, set of reordering-based recipes that satisfy JMM constraints.

As part of our overall effort on certifying the correctness of Java compilers and their associated runtime, we developed the first systematic formalization of the cookbook, providing a semantic basis upon which the relationship between the recipes defined by the cookbook and the guarantees enforced by the JMM can be rigorously established. Notably, one artifact of our investigation is that the rules defined by the cookbook for compiling Java onto the Power multicore microprocessor are *inconsistent* with the requirements of the JMM, a surprising result, and one which justifies our belief in the need for formally provable definitions to reason about sophisticated (and racy) concurrency patterns in Java, and their implementation on modern-day relaxed-memory hardware.

A consequence of our formalization is the ability to mechanize simulation arguments between an architecture-independent intermediate representation of the kind suggested by [49] with machine abstractions for Power and x86. Moreover, our technique enabled a methodology for providing fixes for cookbook recipes that are inconsistent with the behaviors admitted by the target platform, and prove the correctness of these repairs.

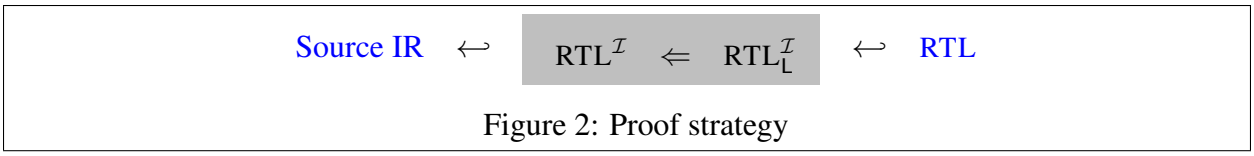

Figure 2: Proof strategy

4 RESULTS AND DISCUSSION

4.1 Proof Methodology

We have developed a new proof methodology to verify the correctness of compiler translations from a high-level intermediate representation with support for object allocation, field access, thread creation and synchronization, and memory management to a low-level structured RTL representation expressed in an IR called $\text{RTL}^{\mathcal{I}}$. The $\text{RTL}^{\mathcal{I}}$ IR is patterned after CompCertTSO's [76] RTL, an IR that expresses unstructured control flow graphs, additionally allowing the expression of high- and low-level statements; these statements are expressed in a structured language called $\mathcal{I}$. An important aspect of $\mathcal{I}$ is its support for coarse-grained atomic instructions, that while not directly available in the target architecture, are only used to support our atomicity refinement proofs. As such, there is a sublanguage of $\mathcal{I}$ which contains all the low-level (fine-grained) statements that are directly supported by the architecture, we denote this language by $\mathcal{I}_{\mathsf{L}}$ (read "Inject Low").

Our new proof methodology is based around an expressive notion of refinement that enables lightweight compositional reasoning of concurrent and potentially racy code within a verified compiler framework. We concentrate on the code that is injected by the compiler to support services such as allocators, collectors, synchronization, etc. Our methodology is integrated within the CompcertTSO verified compiler stack [76]. The refinement technique supports TSO relaxed memory semantics to allow the verification of low-level concurrent code in the context of x86 multiprocessors. We have validated our methodology via the verified compilation of injected concurrent program fragments that interact with a realistic concurrent garbage collector. Figure 2 illustrates our methodology. The shaded portion is enabled via our refinement methodology. $\text{RTL}^{\mathcal{I}}$ programs are successively refined to replace low-level statements with high-level ones based on our refinement rules. $\hookleftarrow$ is the basic backward simulation. $\Leftarrow$ is the backward simulation from refinement.

Figure 1 presents the $\mathcal{I}$ language, with $\mathcal{I}_{\mathsf{L}}$ restricted to the two first lines of the grammar. $\mathcal{I}_{\mathsf{L}}$ has mostly standard commands with the exception that all statements operate on registers, here ranged by the metavariables d, r, o, n and $\vec{r}$ representing a sequence of registers. $\mathcal{I}_{\mathsf{L}}$ includes, skip, sequencing, standard arithmetic and boolean operators, conditionals, repeat$-$until loops, loads-from and stores-to memory (where the registers are assumed to contain memory locations), a compare-and-set statement corresponding to the CAS instruction found on x86 processors, a fence command for memory ordering purposes, and an abort command to denotep exceptional behavior. Notice that the commands $\mathsf{load}_{\nu}(d, r)$ and $\mathsf{store}_{\nu}(d, r)$ have a visibility annotation ν which can be Local or empty. This annotation, which has no runtime effect, indicates in the program syntax that no other thread in the system can modify the references being accessed by the command. More unusual are the "high-level" assume, branch, loop and the coarse-grained atomic statements which complete the $\mathcal{I}$ language. Atomic statements execute disallowing actions from other threads,

loops execute their body a non-deterministic number of times, and branches non-deterministically choose the branch they should execute; "incorrect" choices simply manifest as failed assumptions (expressed through assume) in the resulting execution.

We inject terms of $\mathcal{I}$ on top of the RTL intermediate representation of the CompCertTSO [75] verified compiler. Thus, some nodes of the RTL language of CompCertTSO will contain $\mathcal{I}$ statements. $\text{RTL}^{\mathcal{I}}$ (read "RTL-Inject") is the language resulting from combining RTL with $\mathcal{I}$. The sublanguage that results by combining RTL with the $\mathcal{I}_{\mathsf{L}}$ sublanguage of $\mathcal{I}$ is denoted $\text{RTL}_{\mathsf{L}}^{\mathcal{I}}$.

Given a low-level statement s_l defined as part of the translation, we must construct a high-level statement that matches a provided specification s_h, defined in terms of atomic, assume, loop, branch, and sequence commands; and a proof that s_l refines s_h (written $s_l \preccurlyeq s_h$). s_l is a proper implementation of s_h whenever the visibility annotations of s_l hold. To ease the construction of such proof, we provide a set of compositional rules that can be applied interactively using the Coq proof assistant. These rules avoid the need to modify the semantics of any intermediate representation. We show an excerpt of selected rules provided in our development in Figure 3.

The rule TRANS establishes the obvious transitivity property of refinement. IFBRANCH and REPEAT allow control structures to be replaced by a combination of assume, loop and branch statements. For example, a `repeat` statement can be refined into one that executes its body a non-deterministic number of times, verifying that the terminating condition is not satisfied, and a terminating iteration where the condition is satisfied. IFATOMIC allows an `if` whose branches are atomic to be transformed into an atomic `if`.

The CAS-FAIL rule establishes a refinement between a failed CAS operation and a `load` operation that reads the contents of the location in register r into the destination register d. As in x86-TSO, the load performed by the CAS must be preceded by a fence command. A CAS fails when the presumed old value is not the same as the value read. Thus, the sequence of low-level statements that performs the CAS and then assumes the failing condition is a refinement of a simple load on the location. In contrast, a successful CAS must atomically store the new value into the location, assuming the location still contains the presumed old value (CAS-SUCCESS). Notice that unlike CAS-FAIL, the CAS-SUCCESS rule does not require a fence. This is because the semantics of atomic blocks implicitly requires that the TSO write-buffers be empty, similar to the fence instruction (see subsubsection 4.1.1). Rule SWAPASSUME lifts assumptions above other statements in a sequence. Rule DEAD allows us to remove a statement with an unused effect. This is a typical exercise with racy algorithms: a `while` or `repeat` loop spins until the current thread takes its turn on a shared memory access. By turning such a loop into a mix of loop and assume statements, the last iteration where the thread gets its launching window becomes explicit. The previous iteration block is generally a dead block that can be removed since the actions performed within those iterations have no observable effect. The rule FENCEATOMIC is an obvious consequence of the fencing behavior of atomic that flushes the store buffer upon completion. FENCEELIM allows us to remove unnecessary fences. AFTERABORT indicates that no commands are executed after an abort.

The rule MAKESTOREATOMIC is implied by the fencing behavior of atomic and observing that stores are indivisible operations. A similar argument is applied for MAKELOADATOMIC, but in

$$\textsc{Refl}\ \frac{}{s \preccurlyeq s} \qquad \textsc{Trans}\ \frac{s_1 \preccurlyeq s_2 \quad s_2 \preccurlyeq s_3}{s_1 \preccurlyeq s_3} \qquad \textsc{Repeat}\ \frac{}{\mathsf{repeat}\, s\, \mathsf{until}\, c \preccurlyeq \mathsf{loop}\,(s; \mathsf{assume}\, \neg c\,); s; \mathsf{assume}\, c}$$

$$\textsc{IfBranch}\ \frac{}{\mathsf{if}\, c\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \preccurlyeq \mathsf{branch}\,(\mathsf{assume}\, c\,; s_1),\ (\mathsf{assume}\, \neg c\,; s_2)}$$

$$\textsc{IfAtomic}\ \frac{}{\mathsf{if}\, c\, \mathsf{then}\,(\mathsf{atomic}\, s_0)\, \mathsf{else}\,(\mathsf{atomic}\, s_1) \preccurlyeq \mathsf{atomic}\,(\mathsf{if}\, c\, \mathsf{then}\, s_0\, \mathsf{else}\, s_1)}$$

$$\textsc{CAS-fail}\ \frac{}{\mathsf{cas}(d, r, o, n); \mathsf{assume}\, o \neq d \preccurlyeq \mathsf{fence}; \mathsf{load}\,(d, r)}$$

$$\textsc{CAS-success}\ \frac{}{\mathsf{cas}(d, r, o, n); \mathsf{assume}\, o = d \preccurlyeq \mathsf{atomic}\,(\mathsf{load}\,(d, r); \mathsf{assume}\, o = d\,; \mathsf{store}\,(r, n))}$$

$$\textsc{SwapAssume}\ \frac{\mathsf{defines}(s) \cap \mathsf{uses}(c) = \emptyset}{s; \mathsf{assume}\, c \preccurlyeq \mathsf{assume}\, c\,; s} \qquad \textsc{DeadCode}\ \frac{s_1 \text{ is dead}}{s_1; s_2 \preccurlyeq s_2} \qquad \textsc{FenceAtomic}\ \frac{}{\mathsf{fence} \preccurlyeq \mathsf{atomic}\, \mathsf{skip}} \qquad \textsc{FenceElim}\ \frac{}{\mathsf{fence} \preccurlyeq \mathsf{skip}}$$

$$\textsc{AfterAbort}\ \frac{}{\mathsf{abort}; s \preccurlyeq \mathsf{abort}} \qquad \textsc{MakeStoreAtomic}\ \frac{}{\mathsf{store}(d, r); \mathsf{fence} \preccurlyeq \mathsf{atomic}\, \mathsf{store}(d, r)} \qquad \textsc{MakeLoadAtomic}\ \frac{}{\mathsf{fence}; \mathsf{load}\,(r, d) \preccurlyeq \mathsf{atomic}\, \mathsf{load}\,(r, d)}$$

$$\textsc{GrowAtomicLocal}\ \frac{s_0 \in \{\ \mathsf{store}_{\mathsf{Local}}(d, r), \mathsf{load}_{\mathsf{Local}}(d, r)\ \}}{s_0; \mathsf{atomic}\, s_1 \preccurlyeq \mathsf{atomic}\,(s_0; s_1)} \qquad \textsc{EFLeft}\ \frac{s_1 \text{ is effect free}}{s_1; \mathsf{atomic}\, s_2 \preccurlyeq \mathsf{atomic}\,(s_1; s_2)}$$

$$\textsc{EFRight}\ \frac{s_2 \text{ is effect free}}{\mathsf{atomic}\, s_1; s_2 \preccurlyeq \mathsf{atomic}\,(s_1; s_2)}$$

Figure 3: Compositional rules of the refinement predicate (excerpt).

Arrow	Synchronizes		Meaning
$\xrightarrow{ev}_{\nearrow}$			Single-thread Contribution
$\xrightarrow{ev}_{\searrow}$			TSO Memory Machine
$\xrightarrow{ev}_{t}$	$\xrightarrow{ev}_{\nearrow}$	$\xrightarrow{ev}_{\searrow}$	Memory and Threads Composition (no atomics)
$\rightarrow_{t}$	$\xrightarrow{ev}_{t}$	$\xrightarrow{ev}_{\nearrow}$	Full System Composition
$\overset{tr}{\hookrightarrow}$	$\xrightarrow{ev}_{\searrow}$	$\overset{tr}{\hookrightarrow}$	Abstract Environment Trace
$\overset{tr}{\Rightarrow}$	$\overset{tr}{\hookrightarrow}$	$\overset{tr'}{\Rightarrow}$	Single-thread with Abstract Environment

Figure 4: Synchronization of the Different Semantics

this case the fence is required to precede the load, which in TSO disallows the load from overtaking previously issued writes in the buffer. Perhaps the most interesting rule is GROWATOMICLOCAL which allows local memory operations (i.e., loads and stores) to be moved within an atomic block; such aggregation is clearly acceptable since the effect of the operation is not observable to the environment. This is guaranteed by the Local visibility annotation, which implies that the pointer in the register r cannot be changed by the environment (neither can it be observed in the case of a store). Similar rules EFLEFT and EFRIGHT apply for *effect free* operations (i.e., which only manipulate registers).

Note that the rules shown in Figure 3 are purely syntactical. This helps us reduce the burden of interactively applying them by a set of custom Coq tactics that automatically explore a program tree in order to find a subterm that fits with a given refinement rule. Some rules such as DEAD require discharging some preconditions in order to be applied. We discharge these preconditions using Coq's reflection capabilities; the predicates are executable and we let Coq prove them by computation. Significantly, these rules are sound with respect to the semantics given in Section 4.1.1.

To validate the efficacy of our refinement methodology for the verification of a managed concurrent programming language such as Java, we have devised a block-structured Managed Intermediate Representation (MIR), which we compile to $\text{RTL}_{\mathsf{L}}^{\mathcal{I}}$ and subsequently to x86-TSO using the CompCertTSO tool chain. MIR exposes typical features found in a managed language such as object allocation, field access, synchronization, etc., as well as high-level concurrency primitives such as locks, threads, non-blocking stacks and garbage collection. MIR has been designed to serve as a reasonable IR target for Java bytecodes.

The compiler is sufficiently complete to compile data-allocation intensive programs such as the *binary-trees* benchmark.[1] Running this program shows that the collector effectively traces the heap and collects free objects in parallel with user code.

4.1.1 Formalization

In this section, we present the semantics that justify our methodology. Figure 4 presents the different relations (arrows) we use, and the way in which they synchronize. We start our discussion with the semantics of $\mathcal{I}$. We elide the semantics of $\text{RTL}^{\mathcal{I}}$, which is simply the semantics of the RTL language of CompCertTSO with the additional commands of $\mathcal{I}$. As mentioned before, only terms in $\mathcal{I}_{\mathsf{L}}$, the low level commands of $\mathcal{I}$, are compiled into RTL. Terms in $\mathcal{I}_{\mathsf{H}}$ need not have an obvious implementation in RTL, and only serve to facilitate our proofs.

Our semantics are structured as the composition of different labeled transition systems. Figure 5 presents the events and small-step semantics of individual commands of the $\mathcal{I}$ language. Notice that we have added placeholders $\{\pi\}$, standing for *assertion predicates*, to the syntax of load and store instructions. These predicates will not be used in the definition of the program semantics but are necessary to support the rely-guarantee proof methodology used to aid compositional proof reasoning.

Our labels are composed of memory, synchronization, and error events. Memory events $\mathcal{M}Ev$, roughly correspond to the memory operations available in the x86 architecture. These include: reads $\mathsf{rd}_{p,v}$, representing the query of memory location p which returns value v; writes $\mathsf{st}_{p,v}$, representing the result of a store to a location of a value v found in a memory location p; compare-and-set events $\mathsf{cas}_{p,v,v',w}$, representing an atomic read-modify operation on memory location p where v is the expected value, v' is the value to be stored in p and w is the result of the read – notice that the update is executed only if v and w coincide; an event recording the execution of a memory fence $\#$; and a special event to denote the flush of a TSO buffer $\mathsf{ubff}_{p,v}$. The full set of events Ev includes memory events as well as a τ (empty event) corresponding to a thread-local operation; we omit such labels in general; $\triangleright$ and $\triangleleft$ events, representing the beginning and the end of an atomic command respectively; and an abort event, $\dagger$, generated by the abort command to represent exceptional execution.

The semantics of Figure 5 represents the contribution of each thread, through events, to the overall system. Figure 6 shows the small-step semantics of the composition of different threads and their interaction through shared-memory. Recall that based on the syntax of Figure 1, metavariables $r, o, n, d \in$ *Registers* represent registers, v ranges over values, and p represents a memory location. We distinguish the sublanguage $\mathcal{I}_{\mathsf{L}}$ of $\mathcal{I}$ by disallowing the high-level statements for $\mathcal{I}$ (i.e., assume, loop, branch and atomic).

Thread local evaluation is defined by a small-step evaluation judgment of the form $s,\ rs \xrightarrow{ev}_{\tau} s', rs'$, where s and s' are commands in $\mathcal{I}$, $rs, rs' \in$ *RegMap* represent register maps, associating values to the registers of the thread. We use the command skip to represent termination. The notation $rs[r : v]$ denotes a register map that associates v to the register r. The judgment states that evaluating statement s with a register map rs yields a state with continuation s' and a new register map rs' while emitting the event ev. Notice that when an abort command is executed, the whole command is immediately terminated – with continuation skip and abort event $\dagger$. Since load and compare-and-set judgments are defined in isolation from the memory judgments, but depend on the memory,

[1] http://shootout.alioth.debian.org/

LANGUAGE : $\mathcal{I}$

$$
\begin{array}{rcl}
s \in \mathcal{I} & ::= & \mathsf{skip} \mid d = op(\vec{r}) \mid s; s \mid \mathsf{if}\ c\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{repeat}\ s\ \mathsf{until}\ c \\
& \mid & \{\pi\}\,\mathsf{load}_{\nu}(d, r) \mid \{\pi\}\,\mathsf{store}_{\nu}(d, r) \mid \mathsf{cas}(d, r, o, n) \mid \mathsf{fence} \mid \mathsf{abort} \\
& \mid & \mathsf{atomic}\ s \mid \mathsf{assume}\ c \mid \mathsf{branch}\ s,\ s \mid \mathsf{loop}\ s \\
\nu & ::= & \mathsf{Local} \mid \epsilon
\end{array}
$$

EVENTS : $\mathcal{M}Ev$, Ev

$$
\begin{array}{rcl}
e \in \mathcal{M}Ev & ::= & \mathsf{rd}_{p,v} \mid \mathsf{st}_{p,v} \mid \mathsf{cas}_{p,v,v',w} \mid \mathsf{ubff}_{p,v} \mid \# \\
ev \in Ev & ::= & e \mid \tau \mid \triangleright \mid \triangleleft \mid \dagger
\end{array}
$$

STEP EVALUATION : $(\mathcal{I} \times \mathit{RegMap}) \xrightarrow{ev} (\mathcal{I} \times \mathit{RegMap})$

$$
\begin{array}{rcll}
\mathsf{load}_{\nu}(d, r),\ rs[r : p] & \xrightarrow{\mathsf{rd}_{p,v}} & \mathsf{skip},\ rs[d \leftarrow v] & \\
\mathsf{store}_{\nu}(d, r),\ rs[d : p, r : v] & \xrightarrow{\mathsf{st}_{p,v}} & \mathsf{skip},\ rs & \\
\mathsf{cas}(d, r, o, n),\ rs[r : p,\ o : v, n : v'] & \xrightarrow{\mathsf{cas}_{p,v,v',w}} & \mathsf{skip},\ rs[d \leftarrow w] & \\
d = op(\vec{r}),\ rs & \rightarrow & \mathsf{skip},\ rs[d \leftarrow \mathcal{O}(op, \vec{r})] & \\
\mathsf{skip}; s,\ rs & \rightarrow & s,\ rs & \\
\mathsf{if}\ c\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ rs & \rightarrow & s_1,\ rs & \text{if } \mathcal{C}(c, rs) \\
\mathsf{if}\ c\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ rs & \rightarrow & s_2,\ rs & \text{if } \neg\mathcal{C}(c, rs) \\
\mathsf{repeat}\ s\ \mathsf{until}\ c,\ rs & \rightarrow & \left(\begin{array}{l} s; \mathsf{if}\ c\ \mathsf{then} \\ \quad \mathsf{repeat}\ s\ \mathsf{until}\ c \\ \mathsf{else\ skip} \end{array} \right),\ rs & \\
\mathsf{fence},\ rs & \xrightarrow{\#} & \mathsf{skip},\ rs & \\
\mathsf{loop}\ s,\ rs & \rightarrow & (s; \mathsf{loop}\ s),\ rs & \\
\mathsf{loop}\ s,\ rs & \rightarrow & \mathsf{skip},\ rs & \\
\mathsf{branch}\ s_1,\ s_2,\ rs & \rightarrow & s_i,\ rs & i \in \{1, 2\} \\
\mathsf{assume}\ c,\ rs & \rightarrow & \mathsf{skip},\ rs & \text{if } \mathcal{C}(c, rs) \\
\mathsf{atomic}\ s,\ rs & \xrightarrow{\triangleright} & s; \mathtt{endatomic},\ rs & \\
\mathtt{endatomic},\ rs & \xrightarrow{\triangleleft} & \mathsf{skip},\ rs & \\
\mathsf{abort},\ rs & \xrightarrow{\dagger} & \mathsf{skip},\ rs &
\end{array}
$$

$$
\frac{s_0,\ rs \xrightarrow{ev} s_0',\ rs'}{(s_0; s_1),\ rs \xrightarrow{ev} (s_0'; s_1),\ rs'} \qquad \frac{s_0,\ rs \xrightarrow{\dagger} s_0',\ rs'}{(s_0; s_1),\ rs \xrightarrow{\dagger} \mathsf{skip},\ rs'}
$$

Figure 5: Events and thread-local semantics of $\mathcal{I}$.

their rules are non-deterministic. For example a $\mathsf{rd}_{p,v}$ step must admit every possible value v as its return value. The value is only constrained when synchronizing with the memory, where only one value can be read. The property of accepting all possible return values is called *receptiveness* in CompCertTSO [76], and our semantics uses the same principle.

Statements fence and $\mathsf{cas}(d, r, o, n)$ emit the events $\#$ and $\mathsf{cas}_{p,v,v',w}$ respectively with the obvious semantic rules. For the latter instruction, the memory location to be read-and-modified is contained in the register r. Hence, if the register r contains a pointer p; the expected value for the pointer, given in register o, is v; the value to write, in register n, is v'; and the actual value of p in memory is w, the instruction generates the event: $\mathsf{cas}_{p,v,v',w}$. The value w is placed in the destination register d. In the case where $v = w$, the location p is updated to v', otherwise it remains unchanged. Here also, the rule for $\mathsf{cas}_{p,v,v',w}$ is *receptive*. Rules related to local control flow emit τ events, whose labels we omit since their effect is not observable for other threads.

The command $\mathsf{loop}\, s$ nondeterministically chooses to either execute the statement s and continue looping, or terminate immediately. The statement $\mathsf{branch}\, s_1,\ s_2$ nondeterministically chooses to execute s_1 or s_2. The command $\mathsf{assume}\, c$ only proceeds if register map rs satisfies the condition c. The $\mathsf{atomic}\, s$ command executes s atomically, ensuring that the effect of the atomic action is propagated to memory from the local store buffer upon completion; `endatomic` is a *runtime statement* simply used as a marker to record the end of an atomic section. It is not part of the source code syntax.

In Figure 6 we present the semantics of thread composition stratified into two parts: (1) the semantics of the memory machine, and (2) the overall system behavior composing the memory and the threads. The memory machine implements the TSO memory model following the guidelines of CompCertTSO. The memory state, which shall remain abstract throughout the paper, contains a *store*, mapping memory locations to values, and a *write buffer* for each thread. A write buffer is simply a FIFO queue of store events of the form $\langle p, v\rangle$ (a pending store of a value v at address p). Given a memory $M \in \mathit{Mem}$, we use projections $M.\mathsf{m}$ and $M.\mathsf{b}$ to obtain the store and the buffer map, resp.; $M.\mathsf{b}(t)$ represents the buffer of thread t and the operations $\mathsf{bufferPush}(M, t, \langle p, v\rangle)$, $\mathsf{bufferPop}(M, t)$, $\mathsf{updateMem}(M,\ \langle p, v\rangle)$ and $\mathsf{emptyBuff}$ have the obvious meanings, where M is a memory state, t a thread. $\mathsf{lastIn}(B,\ p)$ returns the value of the last-in item in the store buffer B for the location p. Finally, the operation $\mathsf{CAS}\, M\, p\, v\, v' = (w, M')$ returns the pair containing the value w read in the memory M for pointer p, and accordingly the new memory M' (which will differ from M in case the operation was successful.)

The semantics of the memory machine is described by judgments of the type $M \xrightarrow{ev}_t M'$ which represent the execution of an event ev by thread t and that modifying the memory state M into the state M'. These rules closely follow the memory machines described in [15, 76]; note that in the rule for reading, we use the notation $\mathsf{lastIn}(M.\mathsf{b}(t),\ p)_{M.\mathsf{m}\,(p)}$ to indicate that the absence of location p in the store buffer for thread t – i.e., $p \notin \mathrm{dom}(M.\mathsf{b}(t))$ – results in reading the contents of p from memory: $M.\mathsf{m}\,(p)$. Note that the unbuffering is the only memory operation that is not derived from the program syntax; it can be applied at any time when the thread buffer is non-empty, and flushes some unspecified portion of the buffer to memory.

The state of the whole system is comprised of two components, a global memory, and a thread map (Π), which maps thread identifiers to thread states; these states contain the registers and the code

MEMORY : $\mathit{Mem} \xrightarrow{ev}_{\mathit{Tid}} \mathit{Mem}$ WITH $\mathit{Mem} = ((\mathit{Refs} \rightharpoonup \mathit{Value}) \times (\mathit{Tid} \rightharpoonup \mathit{Buf}))$

$$\frac{v = \mathsf{lastIn}(M.\mathsf{b}(t),\, p)_{M.\mathsf{m}\,(p)}}{M \xrightarrow{\mathsf{rd}_{p,v}}_t M} \qquad \frac{M' = \mathsf{bufferPush}(M, t, \langle p, v\rangle)}{M \xrightarrow{\mathsf{st}_{p,v}}_t M'}$$

$$\frac{\mathsf{CAS}\, M\, p\, v\, v' \;=\; (w, M') \qquad M.\mathsf{b}(t) = \mathsf{emptyBuff}}{M \xrightarrow{\mathsf{cas}_{p,v,v',w}}_t M'} \qquad \frac{M.\mathsf{b}(t) = \mathsf{emptyBuff}}{M \xrightarrow{\#}_t M}$$

$$\frac{\mathsf{bufferPop}(M, t) = (\langle p, v\rangle, M') \neq \bot \qquad M'' = \mathsf{updateMem}(M',\, \langle p, v\rangle)}{M \xrightarrow{\mathsf{ubff}_{p,v}}_t M''}$$

MEMORY COMPOSITION : $(\mathit{Mem} \times \mathit{ThrdSt}) \rightarrow_{\mathit{Tid}} (\mathit{Mem} \times \mathit{ThrdSt})$

MEMORY STEP
$$\frac{st \xrightarrow{ev} st' \qquad M \xrightarrow{ev}_t M'}{(M, st) \xrightarrow{ev}_t (M', st')}$$

INTRA STEP
$$\frac{st \rightharpoonup st'}{(M, st) \rightarrow_t (M, st')}$$

UNBUFFER
$$\frac{M \xrightarrow{\mathsf{ubff}_{p,v}}_t M'}{(M, st) \xrightarrow{\mathsf{ubff}_{p,v}}_t (M', st)}$$

THREAD COMPOSITION : $(\mathit{Mem} \times \mathit{ThrdMap}) \rightarrow_{\mathit{Tid}} (\mathit{Mem} \times \mathit{ThrdMap})$

INTERLEAVE NONATOMIC
$$\frac{(M, \Pi(t)) \xrightarrow{ev}_t (M', st') \qquad ev \neq \triangleright}{(M, \Pi) \rightarrow_t (M', \Pi[t \leftarrow st'])}$$

INTERLEAVE ATOMIC
$$\frac{\Pi(t) \xrightarrow{\triangleright} st' \qquad (M, \Pi[t \leftarrow st']) \rightarrow_t^* (M', \Pi') \qquad M'.\mathsf{b}(t) = \mathsf{emptyBuff} \qquad \Pi'(t) \xrightarrow{\triangleleft} st''}{(M, \Pi) \rightarrow_t (M', \Pi'[t \leftarrow st''])}$$

Figure 6: Memory and Thread Composition Semantics

of the thread. There are two judgments in this semantics. Judgments of the form: $(M, st) \xrightarrow{ev}_t (M', st')$, where st is a the thread state for thread t contains t's continuation and register map, represents the execution of a step by t with respect to shared memory. The rule MEMORY STEP synchronizes the semantics of individual threads and the memory system by having the events in the premises and in the consequent coincide. The rule INTRA STEP does not need to exercise the memory machine, and the rule UNBUFFER asynchronously flushes elements of the buffer into the memory without modifying the thread state.

Thread composition judgments have the shape $(M, \Pi) \rightarrow_t (M', \Pi')$. This semantics captures in a single step, the multiple steps that could be required to execute an atomic statement. The rule INTERLEAVE NONATOMIC executes any statement labelled with an event other than $\triangleright$. The rule INTERLEAVE ATOMIC executes the atomic statement in a single step thereby ensuring that all actions in the atomic statement occur without interleaving of other threads – observe that the thread identifier in the premise restricts the multistep in the premise to only execute steps of thread t.

4.2 Reconciling Language and Processor Memory Models

A decade ago, the semantics of concurrent Java programs, the JMM, was revised and redefined [57]. This revision, which was adopted as part of the official Java specification [46] had multiple purposes. First, it was intended to replace the previous specification which disallowed many common architectural and compiler optimizations of Java programs that were found in many state-of-the-art implementations. Second, it formalized, using a rather complicated axiomatic semantics, the possible behaviors of concurrent Java programs. Its formalization, the *Data Race Freedom* (DRF) guarantee [1], established that programs that do not have data races (i.e., were *data-race free*) in their sequentially consistent (SC) semantics, can only exhibit SC behavior, even when executed on non-SC hardware [7]. Unfortunately, due to the complexity of the formalism, many desirable properties of the semantics were not met, and many undesirable properties were not prevented [74]. In light of these shortcomings, there is an ongoing community effort to better understand and reconsider the definition of the JMM [43].

A testament to the complexity of the JMM specification is the *The JSR-133 Cookbook for Compiler Writers* [49], an informal guide to implementing the JMM in different computer architectures. This document is intended to aid Java compiler writers to provide safe, reasonably efficient implementations, that nonetheless satisfy the JMM requirements. Unlike the JMM, the high-level semantics of Java concurrency is described operationally, in terms of memory instruction reorderings, thus defining the relaxed behaviors a program may exhibit, in a form suitable for reasoning about the correctness of compiler optimizations.

One of the reasons why the current JMM specification is so complex is that it attempts to uniformly capture the set of memory relaxations induced by both relaxed-memory platforms as well as common compiler optimizations deemed necessary to provide performant Java implementations. A recent effort [17] has considered an alternative approach, namely giving a semantics to Java that captures only the relaxations permitted by the TSO memory model found on x86 architectures [63].

One could attempt to implement this flavor of Java in weaker architectures such as on IBM's Power [72] platform, but this is a substantially more challenging exercise; simply retrofitting the TSO-aware semantics developed in [17] for Power would incur a high performance cost, necessitating injection of low-level synchronization operations between normal variable memory accesses to ensure TSO behavior.

The following question thus presents itself: what is the strongest memory model that would be both (1) efficiently implementable – not requiring synchronization at the low level for non-volatile variables – in architectures as relaxed as Power, and (2) yet have a tractable formal semantics amenable to the rigorous proofs needed to demonstrate compiler correctness arguments à la CompcertTSO [76]? As a corollary, we also wished to understand the semantics of current *implementations* of a JVM with respect to the memory model it supports. JVMs ensure their implementations are consistent with the JMM by making conservative decisions on synchronization and shared-memory accesses. Our interest was in determining if there was a middle ground between the behaviors admitted by relaxed-memory architectures and the JMM, which provides a more tractable, perhaps stronger semantics than the JMM, but which nonetheless enables compilers to provide acceptable performance for modern Java applications.

At first glance, it would appear that many of these questions were answered in [49]. However, given that [49] is an informal document, with no clear – let alone formal – semantic definitions, and no guarantees that the rules defined are correct, our research focussed on a methodology to formalize the semantics induced by its "recipes", deriving as an important by-product, a provable validation that some of the minimal guarantees required by the JMM are satisfied. In this sense, our goals were broadly similar to [8], which provides a provably correct compilation strategy of C++11 into Power. However, operating as we do in the Java context, our challenges were substantially different; not only must our formalization cope uniformly with different architectures given the platform agnostic definition of the JMM, but it must also deal explicitly with a number of JMM-specific features such as its support for "roach-motel" reorderings, explicitly established as a requirement of the JMM [57]. These issues make it infeasible to seamlessly transplant the results from approaches like [8]. Unlike [8], we do not provide a concrete compilation strategy – indicating for example that a fence has to be emitted *immediately after* a volatile store – but rather indicate minimal constraints that must be satisfied by any such strategy – for example a fence must exists in between a volatile store and any subsequent memory action –. We did this to allow flexibility to capture systems like Octet [10] where the fences might be added in garbage collection safe points for example. This follows the spirit of [49].

Perhaps surprisingly, the relation between [57] and [49] had not been considered formally before, and notably our results show that the rules implied by [49] for Power were at odds with the requirements of the JMM. Concretely, while working on our proofs we found a counter-example to the DRF requirement of the JMM if the rules of [49] were used for Power. The example in question is the infamous litmus test – reproduced below – considering only *volatile* variables instead of normal variables. In Java, concurrent conflicting accesses to volatile variables are not considered to form a data races. We display the example below with each thread in a column, and we assume that the object o is shared among all threads, with volatile fields v and w. Variables starting with r

are local to each thread.

$o.v = o.w = 0$ & both fields are volatile							
$o.v = 1;$	$\|$	$o.w = 1;$	$\|$	$r0 = o.v;$	$\|$	$r2 = o.w;$
	$\|$		$\|$	$r1 = o.w;$	$\|$	$r3 = o.v;$
Is $r0 = r2 = 1$ & $r1 = r3 = 0$ allowed?							

The behavior in question cannot be produced under a sequentially consistent semantics. However, this behavior is possible in Power [72]. Moreover, inserting `lwsync` Power barriers in between the two reads in the reading threads would not prevent this behavior from happening as documented in [13, 72].[2] Unfortunately, `lwsync` was the barrier of choice recommended by [49] when our work was started to prevent this relaxation.[3] We tried this Java example in a Power 7 machine, and were able to reproduce the erroneous behavior in the two different JVM's we tested[4], indicating that this is not simply a theoretical inconvenience, but a critical dichotomy between desired semantics and implementations. Our discussions with several VM implementors indicated that (a) the cookbook was heavily used as a crucial reference, given the complexity of the official specification, and (b) some implementations were actually aware of the bug noted above, while others were not; given the subtlety and complexity of the JMM, and the lack of consensus among implementors on a proper implementation strategy, the anecdotal evidence made clear that a cookbook-like document is quite necessary, with a provably correct version even more so. To highlight the subtlety of the issues involved, parts of the cookbook were in fact changed [8] in response to advances in the formalization of processor memory models (e.g., [56, 72]), but in the absence of a formal definition, those changes did not remediate the issues noted here.

In light of these issues, we provided the first formalization (operationally) of the semantics of *compiling* concurrency features in Java as described by [49] into the x86 and Power relaxed-memory architectures. Notably, our high-level semantics propagates the relaxations admitted by Power to normal Java variables. Our choice to propagate Power semantics for normal variables into a high-level semantics is motivated by the fact that any stronger semantics at the high-level would impose synchronization operations for normal variables in Power, one of the weakest processor architectures currently available. This would most likely greatly degrade the performance of concurrent Java programs on that platform, which is on the one hand unnecessary given the JMM definition, and on the other hand not required by [49]. We considered this to be a minimal performance requirement for any acceptably efficient implementation of the JMM on Power. Given that Power is one of the weakest architectural memory models yet studied, we view our high-level semantics as an upper bound of how strong a JMM could be, without penalizing weak architectures like Power. [49] uses an intermediate representation to express memory operation reorderings. We formalized this intermediate representation, and proved a simulation argument between source-level programs and programs compiled to this IR, establishing an inclusion property between behaviors allowed by the target architectures (x86 and Power) and this IR. We additionally formalized the

[2]The behavior manifests because `lwsync` imposes no constraints on when the stores performed by the first two threads become visible to the readers.

[3]After our results were published, the cookbook was updated based on our findings.

[4]The example failed on IBM's JVM and Jikes RVM. Similar examples failed in Fiji's realtime JVM implementation on ARM 7.

Table 1: High-level Roach-Motel Semantics Rules

1st Op.\2nd Op.	Normal Load / Store	Volatile Load / Lock	Volatile Store / Unlock
Normal Load / Store			No
Volatile Load / Lock	No	No	No
Volatile Store / Unlock		No	No

different target architectures we considered in the same framework, and when the rules of [49] are correct, proved that they are so. Additionally, we identified the rules that *do not produce correct implementations*, and proposed corrections, which we then proved sufficient to enforce the expected high-level semantics (e.g., volatile variables must exhibit SC semantics). Our findings have been propagated to the current revision of [49]. These results provide the first formalization that relate the high-level semantics of the JMM with low-level architectural implementations as described in [49].

4.2.1 Methodology Details

Consider the requirements of the JMM with respect to the implementation of synchronization operations, and its relation to the rules provided by the cookbook document. A driving principle of the JMM, dubbed the *roach motel semantics* [57], is that increasing the synchronization of a program cannot *add* new observable behaviors to it. The synchronization operations, formally defined in [57], include locking and volatile memory access operations.[5] The roach motel principle implies that all program transformations that increase the *happens-before* [48] relation of the program – which captures the causality relation of a program enforced through its synchronization actions (locks and volatile accesses) – should be allowed by the memory model. Pragmatically, this means that normal memory operations following a volatile write can be reordered before it, since the resulting program imposes additional synchronization not required by the former. Similarly, normal memory operations preceding a volatile read can be reordered after it. An argument similar to the case of volatile writes applies to unlock operations (a monitorexit in Java bytecote), and the same is true for volatile reads with respect to lock operations (monitorenter). These observations justify the first table presented in the cookbook [49], that describes the reorderings possible at the highest-level considered in that document. We reproduce this in Table 1. The table indicates that two operations can be reordered if the cell is empty, and that they cannot if the cell is marked "No"; the first operation is sampled from the rows and the second one from the columns. Data and control dependencies are assumed to be respected by the cookbook tables. Then, for instance two normal memory operations on different references can be freely reordered, but any two synchronization operations cannot.

Intermediate Representation. Before presenting the requirements for the implementation of these operations for a specific architecture, the cookbook introduces an intermediate low-level

[5] Thread creation, termination, and object initialization are also synchronization operations, but they are not relevant for the ideas discussed here.

Table 2: Low-level Cookbook: Barriers Required

1st Op.\2nd Op.	Normal Load	Normal Store	Volatile Load/Lock	Volatile Store/Unlock
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load/Lock	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store/Unlock			StoreLoad	StoreStore

representation in which memory operations are not assumed to have inherent ordering semantics; instead, operation ordering is imposed through the use of additional barrier – or fence – instructions, that guard the kind of reordering permissible between two memory accesses. At this level, volatile memory operations are assumed to be "implemented" using normal memory operations – corresponding to the operations provided by the ISA of the target architectures –, and the ordering constraints of Table 1 have to be enforced rather than assumed. This intermediate representation assumes that there is a different barrier to prevent the reordering of any two kind of memory operations if the barrier is emitted by the code in between these two accesses. For example, two read operations can be prevented from being reordered if a *Load to Load* barrier (LoadLoad) is emitted in between them by the thread. Similar fences exist between stores and loads, loads and stores and two consecutive stores. Table 2 presents the kind of barriers that must be introduced in this intermediate representation to enforce the semantics of Java delineated by Table 1. This is the second table of [49].

Given the lack of a precise semantics for normal load and store instructions, it is difficult to formally establish the correspondence between the high- and low-level versions. A major contribution of our work was the definition of a tractable semantics for these two layers that enables the correctness proof of the rules relating these two tables.

Store-Atomicity Relaxation A limitation of the cookbook document is that the argumentation is made in terms of operation reorderings, which disregards *store-atomicity* – or write-atomicity – which allows write operations to be propagated to different threads at different times, a relaxation permitted by some architectures, including Power and ARM [5, 72]. One could imagine providing a semantics which considers reordering of operations as the only source of relaxations in the style of the TSO, Partial Store Ordering (PSO) and Relaxed Memory Ordering (RMO) [79] memory models. However, this would be insufficient to capture certain important relaxations that are permitted by architectures with weaker memory models; the following example (similar to the example Write-Read Conflict (WRC) of [72]) illustrates this issue.

$$\begin{array}{c} o.f = o'.f = \texttt{NULL} \\ \hline o.f = o' \quad \| \quad (o.f).f = o \quad \| \quad r0 = o'.f; \\ \qquad\quad \| \qquad\qquad\quad \| \quad r1 = r0.f \\ \hline r0 = o \;\&\; r1 = \texttt{NULL}? \end{array} \tag{1}$$

This program has three threads, which share two objects o and o', each with a single field f initially NULL. We assume that the type of the field f is the same as the type of o and o'. In the result

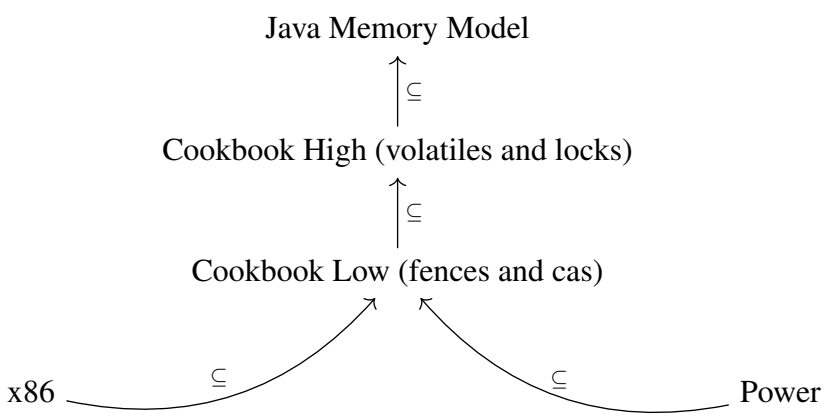

Figure 7: Models above PowerMM exhibit Write-Atomicity Relaxations

indicated at the end, we have that $r0 = o$, therefore it must be the case that the read of $o'.f$ in the third thread returns the object o. Indeed this is possible if the first thread executes first, then the second thread dereferences $o.f$ obtaining o' and after that it writes o into $o'.f$. Now we can fullfil the read of $r0$ in the third thread. It is obvious that the read of $r0.f$ in the third thread cannot happen before $r0$ has obtained its value through the previous read. Therefore these two reads cannot be reordered. In that case, if the only source of relaxation is reordering, the read $r0.f$ which in actuality is a read of $o.f$ must see the value o', since all reorderings are prevented through data dependencies. This final result cannot be produced by a *reordering-only* memory model. However, this is a possible behavior in Power, since a write-atomicity relaxation could mean that the write of the first thread is only propagated to the second, but not the third thread, allowing the third thread to read `NULL` for $r1$. To admit such behavior, it is then necessary to introduce write-atomicity relaxations existent in Power within the (low-level) cookbook semantics to avoid over-synchronizing normal memory accesses.

Proof Structure. Figure 7 illustrates the overall proof structure that we have developed. At the top level, we have the semantics of the JMM as described in [57], or rather the improved version of [74]. Below this level, we have a high-level, architecture-agnostic, operational semantics which adopts Power semantics for normal variables, and sequentially-consistent semantics for volatile variables and locks. We denote this semantics by cookbook-high. One level down, we have the intermediate representation that contains only normal memory accesses and barriers. Finally, at the bottom of the figure we have the semantics of the Power and x86 architectures, of which Power offers a more relaxed semantics. We establish a backwards simulation between the high and low-level definitions of the cookbook, show that high-level cookbook semantics respects the JMM, and that our low-level cookbook definition properly captures the behaviors admitted by x86 and Power.

4.3 Garbage Collection

Modern programming languages like ML, Java, and C# rely on garbage collection (GC) for the automatic reclamation of memory no longer used by the application. The GC is considered to be one of the most subtle parts of modern runtime systems, carefully engineered to minimize runtime overheads of the applications it supports. A family of garbage collection algorithms, named *on-the-fly* garbage collectors [18], allows the detection of garbage and its reclamation to occur concurrently with an application's threads. Such algorithms are notably difficult to implement, test, and prove, and constitute a significant challenge for mechanized verification. Many on-the-fly algorithms are inherently racy, and some algorithms never require application threads (called *mutators*) to wait for the *collector* thread, which detects and frees unused memory. As part of our research goals, we considered the mechanized verification of a state-of-the-art GC algorithm in this landscape [20–22], where no locks are required – *i.e.* it is *lock-free*.

This challenge has been identified and addressed in various settings [31, 32, 35, 36]. Our results provide an independent proof, exploring a different proof method in the design space. First, the backbone of the formalization is a new compiler intermediate representation, named RTIR, that we have developed to implement the garbage collector. Our experience implementing on-the-fly garbage collectors [66] indicates that the choice of programming abstractions is of paramount importance in reasoning and optimizing this kind of algorithm. This concern necessitates a representation that makes the expression and proof of invariants tractable. Moreover, in this work, we strive to make our proof well suited to the context of our larger research goals as described above, aiming at the formal verification of a compiler for concurrent, managed languages.

Our intermediate representation has special support for the implementation of efficient runtime mechanisms: 1. strong type guarantees, 2. abstract concurrent data structures, 3. high-level iterators for reflective inspection of objects used to implement low-level services, *e.g.* ensuring the garbage collector visits every live object 4. native support for threads, and 5. native support for the root management of a concurrent garbage collector (each thread must be able to iterate over the set of memory references it can access directly).

Another important characteristic of our approach is the dedicated rely-guarantee program logic that accompanies our intermediate representation. While previous approaches [31, 32, 36] attack the proof by means of an abstract state transition system requiring a monolithic global invariant be established, we followed the well established rely-guarantee [44] methodology. RG is a major technique for proving the correctness of concurrent programs that provides explicit thread-modular reasoning. In this setting, interferences between threads are described using binary relations: *relies* and *guarantees*. Each thread is proved correct under the assumption it is interleaved with threads fulfilling a *rely* relation. The effect of the thread itself on the shared memory must respect its *guarantee* relation. This guarantee must also be coherent with respect to the relies that the other threads assume. Being able to reason in a thread modular way is key to realize a tractable correctness proof because it avoids the need to explicitly consider all possible interleavings. We prove the soundness of our RG logic, and develop a set of tactics that reduce the proof effort required to discharge the invariants.

Finally, we have developed an original *incremental* proof technique that we put in place to carry

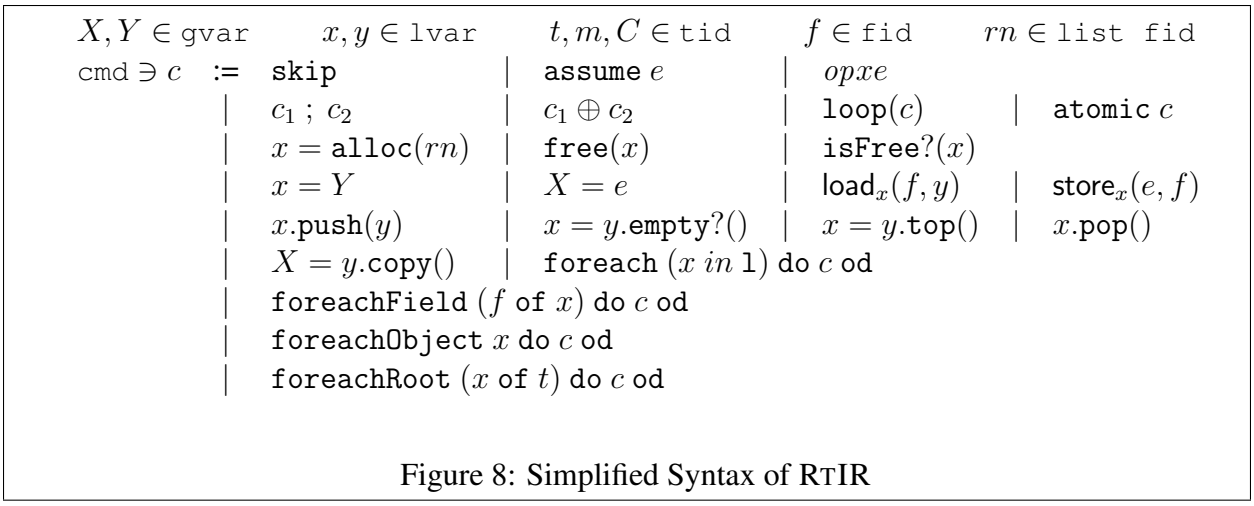

Figure 8: Simplified Syntax of RTIR

out this large endeavor. Starting from the full GC implementation, we progressively annotate the program in order to prove stronger and stronger invariants. At each level, dedicated specification annotations and tactics allow us to refine and reuse what has been proven at the previous levels.

Using the Coq proof assistant, we achieved the following formalizations: 1. the syntax, semantics and the soundness of an RG program logic for our intermediate representation, 2. a number of tactics and structural lemmas to facilitate the so-called *stability proofs* required by the RG methodology, 3. a realistic implementation of Domani *et al.*'s GC algorithm [22] in our intermediate representation and 4. an RG proof ensuring the correctness of the GC: the collector never frees references accessible by the running threads.

4.3.1 The RTIR Intermediate Representation

Syntax. Figure 8 shows the syntax of RTIR. The language provides two kinds of variables: *global* or *shared* variables that can be accessed by all threads, and *local variables* used for thread-local computations. Expressions (e) are built from constants and local variables with the usual arithmetic and boolean operators. Commands include standard instructions, such as `skip`, `assume` e, local variable update $opxe$, and classic combinators: sequencing, non-deterministic choice ($c_1 \oplus c_2$), and loops. The conditional (if e then c_1 else c_2) can be defined as $(\texttt{assume}\ e; c_1) \oplus (\texttt{assume}\ !e; c_2)$, where we write $!e$ for the boolean negation of e. While loops and repeat-until loops can be encoded similarly. RTIR also provides atomic blocks (`atomic` c). In our GC, we use atomic blocks only to add ghost-code – code only used for the proof, not taking part in the computation – and to model linearizable data structures. These atomic constructs can be refined into low-level, fine-grained implementations using techniques such as the atomicity refinement methodology discussed earlier.

Instruction $\texttt{alloc}(rn)$ allocates a new object in the heap by extracting a fresh reference from the freelist – a pool of unused references – and initializing all of its fields in the record name rn to their default value. Conversely, `free` puts a reference back into the freelist. Instruction `isFree?` looks up the freelist to test whether a reference is in it. We use these memory management primitives to implement the GC.

In RTIR, basic instructions related to shared-memory accesses are fine-grained, *i.e.* they perform exactly one global operation (either read or write). These include loads and stores to global variables and field loads and updates. This allows us, when conducting the proofs, to consider each possible interleaving of memory operations arising from different threads, while keeping the semantics reasonably simple. Apart from these basic memory accesses, RTIR provides abstract concurrent queues which implement the *mark buffers* of [22], accessible through standard operations $y = x.\texttt{top}()$, $x.\texttt{pop}()$, $x.\texttt{push}(y)$, $x = y.\texttt{empty?}()$. The use of these buffers are necessary for the implementation of the GC. While we could have implemented these data structures directly in RTIR, we realized that proof burden would be significantly alleviated by higher-level reasoning, and hence to assume that they behave atomically. Mark buffers also provide an operation $X = y.\texttt{copy}()$, to perform a deep copy, only used in ghost code.

A salient ingredient of RTIR is its native support for *iterators*, enabling easy expression of many GC bookkeeping tasks. The iterator `foreach` $(x\ in$ `l`$)$ `do` c `od`, where the variable x can be free in command c, iterates c through all elements x of the static list `l`. Some more sophisticated bookkeeping tasks include the visiting of all the fields of a given object, the marking of each of the *roots* – references bound to local variables – of mutators, or the visiting of every object in the heap (performed during the *sweeping* phase). In those cases, the lists of elements to be iterated upon is not known statically, so we provide dedicated iterators. The iterator `foreachField` $(f$ `of` $x)$ `do` c `od` iterates c on all the fields f of the object stored in x. Command `foreachRoot` $(r$ `of` $t)$ `do` c `od` iterates over the roots of mutator thread t, while `foreachObject` x `do` c `od` iterates over all objects. We stress the fact that iterators have a fine-grained behavior: the body command c executes in a small-step fashion.

Typing information. The semantics of RTIR is enriched with typing information. Basic types in `typ` include `TNum` for numeric constants, `TRef` for references to regular objects (see below), and `TRefSet` for non-null references to abstract mark-buffers. Local variables, global variables, and field identifiers are declared to have exactly one of these types, respectively accessible through functions `lvar_typ`, `gvar_typ` and `fid_typ`. RTIR manipulates two kinds of values: numeric values in the Coq type `Z` and references in `ref`. Types are mapped to values with the function `value` of type `typ -> Type`.

`typ` $\triangleq$ { `TNum`, `TRef`, `TRefSet` }
`lvar` $\triangleq$ `varId` $\times$ `typ`
`gvar` $\triangleq$ `varId` $\times$ `typ`
`fid` $\triangleq$ `fieldId` $\times$ `typ`

```
Definition value (t:typ):Type :=
  match t with
  | TNum => Z
  | TRef | TRefSet => ref end.
```

Execution states Local (resp. global) environments map local (resp. global) variables to values of their declared type. Environments are hence dependent functions of type:

```
Definition lenv := forall x:lvar, value (lvar_typ x).
Definition genv := forall X:gvar, value (gvar_typ X).
```

A thread-local state is defined by a local environment and a command to execute. A global state includes a global environment `ge` and a heap `hp` – a partial map from references to `objects`. We consider two distinct kinds of objects: regular objects, mapping fields to values, and abstract mark-buffers.

```
Definition thread_state := (cmd  lenv).
Record gstate := { ge:  genv; freelist:  ref -> bool;
  hp:  ref -> option object; roots:  tid -> ref -> nat }.
```

Global states also include two components essential to the implementation of a GC: `roots` and a `freelist`. The freelist is indeed a shared data structure, while roots are considered to be thread-local – mutators are responsible for handling their own roots with thread-local counters. Here, we model roots as part of the global state only to ease proof annotations – our final theorem is an invariant of the program global state.

Finally, execution states include the states of all threads and a global state.

```
Definition state := ((tid -> option thread_state)  gstate).
```

Well-typedness invariants

A number of invariants are guaranteed by typing: (i) each variable in the local or global environment contains a value of the appropriate type, (ii) any reference of type `TRef` is either null, in the domain of the heap, or in the freelist, and (iii) each abstract mark-buffer is accessible from a unique global variable, indexed by a thread identifier. This mechanism enforces separation of mark-buffers by typing.

4.3.2 RTIR Proof System

On top of RTIR, we designed a program logic, based on a variation of rely-guarantee, based on our prior experience using this technique for atomicity refinement. When thinking about a particular thread's code, we shall refer to the actions of the other concurrent threads as its *context*. This context is formally encoded as a *rely* relation stating its possible execution steps. Thus, each annotation in the code of a thread must be proved to be *stable w.r.t.* its rely condition, meaning that its validity is not affected by possible state changes induced by any number of rely steps. We follow a similar approach to encode guarantees. In fact, throughout our development we only need to define guarantees, synthesizing the relies of other threads from guarantees.

High-level design choices of proof rules

In our approach, we firstly annotate a program, as is usually done on paper, and then prove the annotated program using syntax-directed proof rules. We thus extend the syntax of commands to include *annotations*. Syntax-directed proof rules were capital for proof automation.

The proof system decouples sequential and concurrent reasoning. Its first layer is a Hoare-like system, with no use of relies or guarantees. A second layer handles interference: proof obligations about relies, guarantees and stability checks of annotations.

Finally, to avoid polluting programs with routine annotations, typically the global invariants, the first layer of the system *assumes* that such invariants hold, and the second layer requires to separately prove their invariance as a stability check.

5 CONCLUSIONS

The three most significant contributions of this project - (1) a compiler infrastructure aware of concurrently executing runtime managed services amenable for formal verification and mechanized proofs; (2) a formalization of the Java cookbook that proves the soundness of compilation schemes from Java source programs to weak memory architectures like Power; and, (3) a fully verified implementation of a concurrent garbage collector built using concepts derived from (1) and (2) validate the thesis underlying the proposed effort. All proofs have been verified in the Coq proof assistant and are publically available.

6 Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, (ISCA 1990), Seattle, WA, June 1990*, pages 2–14, 1990.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *ISCA*, pages 2–14, 1990.

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion. In *Computer Aided Verification, (CAV 2014), Vienna, Austria, July 18-22, 2014. Proceedings*, pages 508–524, 2014.

[4] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pages 141–157, 2013.

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7, 2014.

[6] Arvind and Jan-Willem Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA*, pages 29–40, 2006.

[7] David Aspinall and Jaroslav Sevcík. Formalising Java's Data Race Free Guarantee. In *Theorem Proving in Higher Order Logics, 20th International Conference, (TPHOLs 2007), Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 22–37, 2007.

[8] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2012), Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 509–520, 2012.

[9] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *POPL*, pages 55–66, 2011.

[10] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: Capturing and Controlling Cross-thread Dependences Efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, (OOPSLA 2013), Indianapolis, IN, USA, October 26-31, 2013*, pages 693–712, 2013.

[11] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In *ESOP*, pages 533–553, 2013.

[12] Gérard Boudol and Gustavo Petri. Relaxed Memory Models: an Operational Approach. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2009), Savannah, GA, USA, January 21-23, 2009*, pages 392–403, 2009.

[13] Gérard Boudol, Gustavo Petri, and Bernard P. Serpette. Relaxed Operational Semantics of Concurrent Programming Languages. In *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, (EXPRESS/SOS 2012), Newcastle upon Tyne, UK, September 3, 2012.*, pages 19–33, 2012.

[14] Stephen D. Brookes. Full abstraction for a shared variable parallel language. In *LICS*, pages 98–109, 1993.

[15] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the tso memory model. In *ESOP*, pages 87–107, 2012.

[16] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Programming Languages and Systems, 16th European Symposium on Programming, (ESOP 2007), Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 331–346, 2007.

[17] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. Plan B: a buffered memory model for java. In *POPL '13*, pages 329–342, 2013.

[18] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[19] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, pages 363–377, 2009.

[20] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proc. of POPL 1994*, pages 70–83, 1994.

[21] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of POPL 1993*, pages 113–123, 1993.

[22] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanover. Implementing an on-the-fly garbage collector for Java. In *Proc. of ISMM 2000*, pages 155–166, 2000.

[23] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanover. Implementing an On-the-Fly Garbage Collector for Java. In *ISMM*, pages 155–166, 2000.

[24] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proc. of POPL 2009*, pages 2–15, 2009.

[25] Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, pages 296–311, 2010.

[26] Y. Zakowski et al. Verifying a concurrent garbage collector using an RG methodology, 2017. http://www.irisa.fr/celtique/ext/cgc/.

[27] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. A Syntactic Theory of Sequential Control. *Theor. Comput. Sci.*, 52:205–237, 1987.

[28] Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[29] Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimizations. In *The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 1995), San Francisco, California, USA, January 23-25, 1995*, pages 209–220, 1995.

[30] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, (PLDI 1993), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

[31] P. Gammie, A. L. Hosking, and K. Engelhardt. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In *Proc. of PLDI 2015*, pages 99–109, 2015.

[32] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Proc . of CAV'96*, pages 462–465, 1996.

[33] Petri Gustavo. *Operational Semantics for Relaxed Memory Models*. PhD thesis, Univeristé de Nice Sophia Anitipolis, 2010.

[34] K. Havelund. Mechanical verification of a garbage collector. In *Proc. of IPPS/SPDP'99*, pages 1258–1283, 1999.

[35] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *Proc of POPL 2009*, pages 441–453, 2009.

[36] C Hawblitzel, E. Petrank, S. Qadeer, and S Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proc. of CAV 2015*, 2015.

[37] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proc. of MSP 2002*, pages 256–263, 2002.

[38] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[39] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[40] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, pages 353–367, 2008.

[41] Marieke Huisman and Gustavo Petri. The Java Memory Model: a Formal Explanation. *Verification and Analysis of Multi-threaded Java-like Programs (VAMP'07)*, 2007.

[42] Radha Jagadeesan, Corin Pitcher, and James Riely. Generative Operational Semantics for Relaxed Memory Models. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 307–326, 2010.

[43] JMM Mailing list: Developing the JEP 188: Java Memory Model Update. http://mail.openjdk.java.net/mailman/listinfo/jmm-dev, 2014.

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

[45] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[46] Java Memory Model and Thread Specification, 2004.

[47] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proc. of POPL'15*, pages 637–650. ACM, 2015.

[48] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[49] Doug Lea. The JSR-133 Cookbook for Compiler Writers. http://g.oswego.edu/dl/jmm/cookbook.html.

[50] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[51] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012.

[52] H. Liang, X. Feng, and M. Fu. Rely-Guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Trans. Program. Lang. Syst.*, 36(1):3:1–3:55, 2014.

[53] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, pages 455–468, 2012.

[54] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. Technical report, University of Science and Technology, 2012.

[55] Andreas Lochbihler. Verifying a compiler for java threads. In *ESOP*, pages 427–447, 2010.

[56] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification - 24th International Conference, (CAV 2012), Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 495–512, 2012.

[57] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Special POPL Issue (DRAFT)*, 2005.

[58] 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 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391, 2005.

[59] Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A Case for an SC-preserving Compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI 2011), San Jose, CA, USA, June 4-8, 2011*, pages 199–210, 2011.

[60] A. McCreight, T. Chevalier, and A. P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proc. of ICFP 2010*, pages 273–284, 2010.

[61] M. O. Myreen. Reusable Verification of a Copying Collector. In *VSTTE '10*, 2010.

[62] P. W. O'Hearn. Separation logic and concurrent resource management. In *Proc. of ISMM 2007*, page 1, 2007.

[63] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, (TPHOLs 2009), Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009.

[64] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. Cooking the Books: Formalizing JMM Implementation Recipes (Extended Version), 2015. https://www.cs.purdue.edu/homes/gpetri/publis/CtB-long.pdf.

[65] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. of PLDI 1988*, pages 199–208, 1988.

[66] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proc. of PLDI*, 2010.

[67] PowerPC ISA. Version 2.06 Revision B. IBM, 2010.

[68] L. Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In *Proc. of ESOP 2003*, pages 348–362, 2003.

[69] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS 2002*, pages 55–74, 2002.

[70] Tom Ridge. A rely-guarantee proof system for x86-tso. In *VSTTE*, pages 55–70, 2010.

[71] Vijay A. Saraswat, Radha Jagadeesan, Maged M. Michael, and Christoph von Praun. A Theory of Memory Models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 161–172, 2007.

[72] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER Multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI 2011), San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011.

[73] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proc. of PLDI'15*, pages 77–87. ACM, 2015.

[74] Jaroslav Sevcik and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pages 27–51, 2008.

[75] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, pages 43–54, 2011.

[76] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[77] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[78] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi. End-to-end Sequential Consistency. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 524–535, 2012.

[79] SPARC Corporation. *The SPARC Architecture Manual (V. 9)*. Prentice-Hall, Inc., 1994.

[80] R. Kent Treiber. Systems Programming: Coping with Parallelism - RJ 5118. Technical report, IBM Almaden Research Center, 1986.

[81] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proc. of ICFP'13*, pages 377–390, 2013.

[82] Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL*, pages 247–258, 2011.

[83] V. Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276, 2011.

[84] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. of CONCUR 2007*, pages 256–271, 2007.

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

[86] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying Fence Elimination Optimisations. In *Static Analysis - 18th International Symposium, (SAS 2011), Venice, Italy, September 14-16, 2011. Proceedings*, pages 146–162, 2011.

[87] Y. Zakowski, D. Cachera, D. Demange, and D. Pichardie. Compilation of linearizable data structures - a mechanised RG logic for semantic refinement, 2017. Technical Report, available at https://hal.archives-ouvertes.fr/hal-01538128.

7 List of Symbols, Abbreviations, and Acronyms

CAS - Compare and Set
DRF - Data-Race Freedom
GC - Garbage Collection
IRIW - Independent Reads of Independent Writes
IR - Intermediate Representation
JMM - Java Memory Model
JVM - Java Virtual Machine
MIR - Managed Intermediate Representation
PSO - Partial Store Ordering
RG - Rely-Guarantee
RMO - Relaxed Memory Ordering
SC - Sequentially Consistent
TSO - Total Store Ordering
WRC - Write-Read Conflict