Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations

LENNARD GÄHER, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
SIMON SPIES, MPI-SWS, Germany
RALF JUNG, MPI-SWS, Germany
HOANG-HAI DANG, MPI-SWS, Germany
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
JEEHOON KANG, KAIST, Korea
DEREK DREYER, MPI-SWS, Germany

Today’s compilers employ a variety of non-trivial optimizations to achieve good performance. One key trick compilers use to justify transformations of concurrent programs is to assume that the source program has no data races: if it does, they cause the program to have undefined behavior (UB) and give the compiler free rein. However, verifying correctness of optimizations that exploit this assumption is a non-trivial problem. In particular, prior work either has not proven that such optimizations preserve program termination (particularly non-obvious when considering optimizations that move instructions out of loop bodies), or has treated all synchronization operations as external functions (losing the ability to reorder instructions around them).

In this work we present Simuliris, the first simulation technique to establish termination preservation (under a fair scheduler) for a range of concurrent program transformations that exploit UB in the source language. Simuliris is based on the idea of using ownership to reason modularly about the assumptions the compiler makes about programs with well-defined behavior. This brings the benefits of concurrent separation logics to the space of verifying program transformations: we can combine powerful reasoning techniques such as framing and coinduction to perform thread-local proofs of non-trivial concurrent program optimizations. Simuliris is built on a (non-step-indexed) variant of the Coq-based Iris framework, and is thus not tied to a particular language. In addition to demonstrating the effectiveness of Simuliris on standard compiler optimizations involving data race UB, we also instantiate it with Jung et al.’s Stacked Borrows semantics for Rust and generalize their proofs of interesting type-based aliasing optimizations to account for concurrency.

1 INTRODUCTION

Modern compilers use many non-trivial optimizations to make programs run faster. As an example, consider the following simple function, which sums up the value of \( \ast y \) until a counter reaches \( \ast x \):

```plaintext
1 int foo(int *x, int *y) {
2    int i = 0; int sum = *y;
3    while (i != *x) {
4        i += 1; sum += *y;
5    }
6    return sum;
7 }
```

When passed the -O2 flag, Clang optimizes the body of this function to roughly the equivalent of the code on the right. The compiler has moved all the pointer loads out of the loop, so it can access memory once and then keep the values in registers.

When passed the -O2 flag, Clang optimizes the body of this function to roughly the equivalent of the code on the right. The compiler has moved all the pointer loads out of the loop, so it can access memory once and then keep the values in registers.

Why is this transformation correct? In a sequential language, this is rather trivial, but if concurrency needs to be considered then this optimization might seem incorrect: if another thread changes

\[ \text{It then actually goes further and replaces the entire loop by a multiplication.} \]
\(x\) (or \(y\)) while the loop is running, the original program will use the updated value, but the optimized program will keep using the outdated value! The reason this optimization is correct in languages like C, C++, and Rust is that unsynchronized concurrent accesses (usually called data races) are undefined behavior (UB), which means the compiler may assume that no such accesses happen. Under this assumption, no other thread may write to \(x\) or \(y\), thus validating the optimization.

**Fair termination preservation.** But what does it even mean for this optimization to be “correct”? The obvious aspect of this is refinement of the final result(s) computed by the program: whenever the optimized program terminates with a result \(v\), then that should also be a possible result of the original, unoptimized program.\(^2\)

However, this is not enough: a compiler should also be required to preserve termination of the original program; introducing an infinite loop is not a “correct” transformation. To state the termination-preservation property for concurrent programs, we need to be careful: if the scheduler starves a thread that holds some lock by never letting it take any more steps, it is often easy to construct diverging executions—but these executions are unrealistic; their mere existence should not give the compiler license to introduce infinite loops. So we only wish to consider infinite executions that exhibit a fair schedule, where no thread is left starving. Fair termination preservation demands that if the optimized program has such an infinite execution, then so does the source program.

Finally, optimizations typically happen locally in some function without knowing the larger program they are a part of. We hence need the optimization to be correct no matter the context that this function might be used in.

We are thus aiming to establish fair termination-preserving contextual refinement.

**Correctness of program transformations.** There is a lot of prior work on verifying program transformations, which we briefly discuss to be able to explain the new aspects our work brings to this space. An overview can be found in Figure 1.

When it comes to verifying program optimizations, the most obvious points of comparison are verified optimizing compilers such as CompCert [Leroy 2006, 2009] and CakeML [Kumar et al. 2014]. Both of these flagship projects have verified correctness of a number of non-trivial optimization passes. However, CakeML considers only sequential programs, so data races are not considered.

CompCert has seen many extensions with support for concurrency. CompCertTSO [Ševčík et al. 2013] uses the TSO (total store order) memory model, which does not treat data races as UB, thus ruling out optimizations such as the one above. CASCompCert [Jiang et al. 2019] enriches the sequential semantics with a notion of “footprints” such that correctness of optimizations on the sequential language implies correctness in a concurrent context. A similar approach is also used by the “Concurrent CompCert” line of work [Beringer et al. 2014; Cuellar 2020]. This approach can in principle handle our motivating example, though CompCert does not actually perform such transformations. However, it requires hiding all synchronizing operations behind external function calls (“FFI”) in the sequential semantics; this rules out most optimizations that reorder instructions around such synchronizing operations, such as the one we consider in §3.2. The same applies to CCAL [Gu et al. 2018], which comes with a thread-safe variant of CompCertX [Gu et al. 2015]: while fair scheduling is accounted for there, all accesses to shared memory (not just synchronizing operations) are treated as external function calls and thus not subject to optimizations.

The first work to verify transformations which leverage undefined behavior of data races is found in Ševčík’s PhD thesis [Ševčík 2009, 2011]. However, Ševčík only considers finite traces, and

\(^2\) We are considering languages with non-determinism here, so there might be more than one possible result, and the optimized program might have a subset of the possible results of the original program.
1.1 Simuliris

In this paper, we introduce Simuliris, the first simulation relation to establish fair termination-preserving contextual refinement for concurrent program transformations that can exploit UB.

Iris-style ownership in a coinductive simulation relation. The key idea behind our approach is to leverage the concept of ownership: we formulate our simulation relation in a separation logic. When applied to the motivating example, our proof rules establish that after the load operation \(*x\), we own the underlying memory. We can hold on to this ownership throughout the entire loop, which lets us prove that each time the original program accesses \(*x\), the same result will be returned.

The basic idea of combining separation logic and a simulation relation has been explored before, but prior work has mostly focused on verifying that an implementation of an abstract data type implements a specification. As such, this line of work lacks the ability to exploit undefined behavior in the way we require for our example (as reflected in Figure 1, and discussed further in §8).

Several of these prior publications explore the use of Iris [Jung et al. 2015, 2018b], a framework for defining concurrent separation logics with a flexible form of "ghost ownership", for the purpose of refinement proofs. However, the use of step-indexing [Appel and McAllester 2001] means that Iris-based approaches like ReLoC [Frumin et al. 2018] do not support reasoning about liveness properties such as termination preservation. This limitation can be bent to some extent (see §8), but those approaches have not been shown to apply to concurrent programs with unbounded non-determinism.

Simuliris is based on Iris, and as such inherits its flexible notion of ghost state, which forms the foundation for all our ownership reasoning. However, Simuliris is not step-indexed: verification of program transformations requires different tools than general program verification, and we found that the powerful reasoning principles enabled by step-indexing are not required for our task. Simuliris demonstrates how the heart of Iris, its flexible model of ghost state, can be married as such does not establish termination preservation. The same limitation applies to later work on optimizations for C(-like) memory models [Morisset et al. 2013; Vafeiadis et al. 2015].
together with the typical shape of a coinductive simulation to obtain a simulation relation that supports both liveness properties and powerful ownership-based reasoning.

We also inherit the Iris Proof Mode [Krebbers et al. 2017b, 2018], which lets us carry out interactive proofs in the Coq proof assistant for all results presented in this paper (metatheory and examples). The Coq proofs are available in the supplementary material [Anonymous 2021].

**Fairness and implicit stuttering.** One usually rather tedious aspect of termination-preserving simulation relations is stuttering. In Simuliris, we manage to completely hide the bookkeeping that is usually associated with stuttering by using a technique we call *implicit stuttering*.

Stuttering is required whenever a lock-step simulation between source and target (i.e., unoptimized and optimized program) is insufficient. For example, in the optimization shown above, when the optimized program performs the $^\star x$ before the loop, this does not directly correspond to any step in the source program—so ideally we could just ignore the source when reasoning about this part of the optimized program. However, we have to be careful not to violate termination preservation: if we ignore the source infinitely often, we could end up with a diverging execution in the optimized program even though the source always terminates!

The typical solution to this problem is to add a "stutter counter/metric" that keeps track of how many more steps the target may make before a source step is required. This additional bookkeeping is burdensome and makes it hard to give modular specifications. Instead, we define stuttering *implicitly*, with a least fixed-point instead of an explicit decreasing metric [Spies et al. 2021]. This approach interacts with fairness in non-trivial ways, so we had to develop a new soundness proof to establish that our simulation relation indeed ensures *fair* termination preservation.

**Simuliris instances: data races, Stacked Borrows.** Like Iris, Simuliris is a language-generic framework. The core simulation relation can be instantiated with multiple different programming languages, providing them with many key proof rules (such as the frame rule from separation logic), a parametric coinduction principle, and an adequacy theorem for fair termination-preserving whole-program refinement “for free”. In this paper, we instantiate Simuliris with two different languages.

The first language, SimuLang, is used to demonstrate our ability to perform optimizations based on UB of data races. SimuLang uses the memory model of $\lambda_{Rust}$ [Jung et al. 2018a], which ensures that programs with data races get stuck. We then perform all our proofs under the assumption that the source program will *not* get stuck. This permits us to prove correctness of optimizations such as our motivating example by exploiting the absence of data races for unsynchronized accesses: threads fully *own* locations that they perform non-atomic accesses on, hiding the complexity of reasoning about traces of memory events. This layer of abstraction keeps the formal argument fairly close to the intuitive idea of why such an optimization is correct. The language-independent Simuliris infrastructure (in particular framing and coinduction) makes it easy to move from proving individual reorderings to loop hoisting optimizations such as our motivating example.

Our second language is Stacked Borrows [Jung et al. 2020], a recently proposed aliasing model for Rust that supports strong intraprocedural optimizations based on alias information derived from the Rust type system. Originally, correctness of these optimizations was proven in a coinductive ownership-based simulation relation not unlike the model of Simuliris—but without support for concurrency, implicit stuttering, general Iris-style ghost state, or even a separation logic to abstract away resource ownership. Using Simuliris, we verify correctness of the same optimizations as the original paper, but for a new *concurrent* version of Stacked Borrows. We also establish correctness of a new loop hoisting optimization based on Stacked Borrows.
**Paper structure.** The remainder of the paper is structured as follows: we first give a tour of how Simuliris works in general (§2) and how this approach scales to exploiting undefined behavior of data races (§3). Then we come to the technical meat underlying the framework: we define our notion of fair termination-preserving contextual refinement (§4), we explain the definition of our underlying simulation relation and how it can be used to establish contextual refinement (§5), and we show how the proof rules for exploiting data races are justified (§6). Finally, we briefly explain how Simuliris can be applied to Stacked Borrows (§7), before concluding with related work (§8).

## 2 SIMULIRIS BY EXAMPLE: THE BASICS

In this section, we introduce Simuliris’s core reasoning principles and show how ownership reasoning helps us to prove program optimizations. We start with a very brief explanation of our setup, and then explain the key rules of Simuliris with a series of simple examples. We illustrate how to use local ownership (§2.1), how to interface with unknown code (§2.2), and how to exploit undefined behavior in the original program (§2.3). We then conclude with an optimization involving loops (§2.4), illustrating that our ownership reasoning nicely composes with coinduction.

At the core of Simuliris lies a coinductive simulation relation defined in separation logic. We write this simulation relation in the style of Relational Hoare Logic [Benton 2004]:

\[
\{ P \} e_1 \leq e_2 \{ v_1, v_2, \Phi \}
\]

The idea of these quadruples is that under precondition \( P \), the source expression \( e_1 \) simulates the target expression \( e_2 \) such that both terminate in values related by postcondition \( \Phi \), or both diverge.

Here, \( e_1 \) and \( e_2 \) are terms in SimuLang, our main example language for this paper. An excerpt of the grammar of SimuLang is shown in Figure 2 (omitting standard features such as sums and products). SimuLang is a simple expression-based language with ML-style references, fork-based concurrency, and function pointers. Evaluation order proceeds right-to-left, as determined by the definition of evaluation contexts \( K \). Further details can be found in our appendix [Anonymous 2021, §2].

SimuLang does not have \( \lambda \)-terms; one can think of it as a “post-closure-conversion” language. Correspondingly, function pointers simply consist of the name of a function, and a whole SimuLang program \( \rho \) is a list of mutually recursive function declarations \( f \ x \equ e \) (we implicitly assume that no function name is declared more than once). Any call to this function will reduce to \( e \) with \( x \) substituted by the call argument. Local variables inside a function are bound via \texttt{let}. These variables are immutable; to model mutable stack-allocated variables, we use \texttt{ref}(.). In other words, we do not distinguish between stack-allocated and heap-allocated variables (following prior languages designed for the verification of Rust or C code [Jung et al. 2018a; Sammler et al. 2021]). We use \( e_1 ; e_2 \) as sugar for \texttt{let} \_ \(:= e_1 \text{ in } e_2 \). SimuLang features \texttt{while} loops: the term \texttt{while } \( e_1 \text{ do } e_2 \text{ od} \) reduces to \( \text{if } e_1 \text{ then } (e_2 ; \text{while } e_1 \text{ do } e_2 \text{ od} ) \text{ else } () \) (and there is no evaluation context for loops).

In the following, when writing concrete programs, we will use sans-serif font for typesetting concrete names of program variables or functions (represented as strings in the formal Coq development), while using typical \textit{italic} font for logical variables.

### 2.1 Optimizations on Local Memory Locations

To explain how we use ownership reasoning for compiler optimizations, let us consider a simple example: removing a load from a local (unescape) memory location. In SimuLang, the expression \texttt{let y := ref(42) in ly} allocates a fresh location and then reads from it\(^3\). We would like to optimize the load of \( \text{ly} \) away and directly return 42. This optimization is correct because the value stored in

\(^3\)In order to simplify the presentation, many examples in this section leak memory by omitting deallocations. Of course, our framework supports the same optimizations in the presence of proper deallocation.


\[
\text{Expr} \triangleright \exists e \ ::= \ v \mid x \mid \text{let } x := e_1 \text{ in } e_2 \mid e_1 + e_2 \mid \text{call } e_1 e_2 \mid \text{fork}\{e\}
\]

\[
\mid \text{ref}(e) \mid \text{le } e_1 \leftrightarrow e_2 \mid \text{free}(e) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{while } e_1 \text{ do } e_2 \text{ od} \mid \ldots
\]

\[
\text{Val} \triangleright \exists v \ ::= \ z : \mathbb{Z} \mid b : \mathbb{B} \mid t : \text{Loc} \mid (v_1, v_2) \mid f : \text{FnName} \mid () \mid \ldots
\]

\[
\text{Prog} \triangleright \rho \ ::= (f \ x \ @ e), \rho \mid \emptyset
\]

\[
\text{Ectx} \triangleright K \ ::= \bullet \mid \text{let } x := K \text{ in } e_2 \mid e_1 + K \mid K + v \mid \text{call } e_1 K \mid \text{call } K v
\]

\[
\mid \text{ref}(K) \mid ![K] \mid e_1 \leftarrow K \mid K \leftarrow v \mid \text{free}(K) \mid \text{if } K \text{ then } e_2 \text{ else } e_3 \mid \ldots
\]

Fig. 2. Excerpt of the grammar of SimulLang.

\[
\{\text{True}\} \text{ let } y := \text{ref}(42) \text{ in } 42 \leq \text{let } y := \text{ref}(42) \text{ in } !y \{v, v_3.v_3 = v_5\}
\]

Under the trivial precondition, we can execute the optimized (left, target) and the original (right, source) program, resulting in the same value.

To prove such quadruples, our simulation relation allows us to “focus” on the execution of source and target individually by switching to special source and target \textit{triples}, with rules like \textsc{source-focus} (see Figure 3; the target rule is symmetric). As we will see below, most of the rules for these triples are symmetric and reminiscent of a unary separation logic. With these triples, we can focus on a subexpression of either the source or the target.

The focusing rules combine sequencing and \textit{implicit} stuttering: we can focus on an expression in evaluation position \textit{i.e.}, which is contained in an evaluation context \(K\) on one side and then show that that expression terminates in a value\(^4\) satisfying \(\Psi\) (it must not diverge). Meanwhile, the other side of the program “stutters”, not making any progress. Afterwards, the postcondition \(\Psi\) can be used to continue with the rest of the simulation. (The reader might wonder at this point how stuttering without further side-conditions does not break termination preservation. We will explain this in §5.)

With the focusing triples in hand, we can start the proof. First, we focus on \text{ref}(42) in the source with \textsc{source-focus} to allocate a memory location, which is done by applying \textsc{source-alloc}. As mentioned above, we obtain \textit{local} ownership of this memory location—no other parts of the program can know about it.\(^5\) In ordinary separation logic, this notion of ownership can be expressed using the points-to connective \(\ell \mapsto v\), stating both the knowledge that \(\ell\) contains value \(v\) and exclusive ownership of location \(\ell\). As Simuliris is a \textit{relational} separation logic [Yang 2007], it has two points-to connectives: \(\ell \mapsto^\text{ref} v\) for locations in the target, and \(\ell \mapsto^\text{src} v\) for locations in the source. \textsc{source-alloc} thus picks a fresh location \(\ell_\delta\) and provides the corresponding local ownership of this source location.

Having reduced the source expression, we are left with the goal \(\{\ell_\delta \mapsto^\text{src} 42\} \text{ let } y := \text{ref}(42) \text{ in } 42 \leq \text{let } y := \ell_\delta \text{ in } !y \{v_3.v_3 = v_5\}\). To allocate the reference in the target, we follow the same steps as in the source: we first apply the target focusing rule (which works exactly like source focusing), and then use \textsc{target-alloc}. (Ownership of \(\ell_\delta \mapsto^\text{src} 42\) is maintained using the standard frame rule, which

---

\(^4\)As we will see in §2.4, our full system also supports leaving the focus early with an expression-based postcondition.

\(^5\)In addition, allocation provides us with ownership of assertions about the size of allocations, which are relevant for deallocation and pointer arithmetic. For simplicity, we omit this detail and refer to the supplementary material.
Language-independent rules:

\[
\text{SOURCE-FOCUS} \quad \{ P \} e_s \{ v_s, \Psi v_s \}_{s} \quad \forall v_s. \{ \Psi v_s \} e_t \leq K_s[v_s] \{ \Phi \} \\
\text{SIM-FRAME} \quad \{ P \} e_t \leq e_s \{ \Phi \} \\
\text{SIM-CALL} \quad \{ \forall v_s v_t \} \text{ call } v_t \leq \text{ call } v_s \{ v_t, v_s \} \{ \forall v_s v_t \} \\
\text{SIM-BIND} \quad \{ P \} e_t \leq e_s \{ v_t, v_s, \Psi v_t v_s \} \quad \forall v_t, v_s. \{ \Psi v_t v_s \} K_t[v_t] \leq K_s[v_s] \{ \Phi \} \\
\text{SIM-VALUE} \quad \{ \Phi \} v_t v_s \leq v_s \{ \Phi \} \\
\]

SimuLang rules:

\[
\text{SOURCE-ALLOC} \quad \{ \text{True} \} \text{ ref}(v_s) \{ v_t', \exists_\ell v_s' = \ell * \ell_s \rightarrow v_s' \}_{s'} \quad \text{TARGET-ALLOC} \quad \{ \text{True} \} \text{ ref}(v_t) \{ v_t', \exists_\ell v_t' = \ell * \ell_t \rightarrow v_t' \}_{t'} \\
\text{SOURCE-PURE} \quad e_t \rightarrow \text{ pure } e_t' \quad \{ P \} e_t \leq e_s' \{ \Phi \} \\
\text{TARGET-PURE} \quad e_t \rightarrow \text{ pure } e_t' \quad \{ P \} e_t \leq e_s \{ \Phi \} \\
\text{SOURCE-LOAD} \quad \{ \ell_s \rightarrow v_s \}_{s} \{ v_t', v_s' = v_s * \ell_s \rightarrow v_s' \}_{s'} \\
\text{TARGET-LOAD} \quad \{ \ell_t \rightarrow v_t \}_{t} \{ v_t', v_t' = v_t * \ell_t \rightarrow v_t' \}_{t'} \\
\text{SOURCE-STORE} \quad \{ \ell_s \rightarrow_{-} \}_{s} \ell_s \leftarrow v_s \{ v_t', v_s' = ( * ) \rightarrow_{s} v_s' \}_{s'} \\
\text{TARGET-STORE} \quad \{ \ell_t \rightarrow_{t} \}_{t} \ell_t \leftarrow v_t \{ v_t', v_t' = ( * ) \rightarrow_{t} v_t' \}_{t'} \\
\]

Fig. 3. Core Simuliris and SimuLang rules.

we only show for the relational case: \textsc{sim-frame}.) To then execute \text{ref}(42), we use \text{target-alloc}. We are left to prove \{ \ell_t \rightarrow_{t} 42 * \ell_s \rightarrow_{s} 42 \} \text{ let } y := \ell_t \text{ in } 42 \leq \text{ let } y := \ell_s \text{ in } !\ell_s \{ v_t, v_s, v_t = v_s \}.

Next, we reduce the let-expression in source and target with the pure execution rules \text{source-pure} and \text{target-pure}. We now end up with the remaining goal \{ \ell_t \rightarrow_{t} 42 * \ell_s \rightarrow_{s} 42 \} \leq \{ \ell_s \{ v_t, v_s, v_t = v_s \} \}. At this point, we make a source execution step that is not mirrored in the target (again using our focusing triples): we load 42 from \ell_t. To do so, it is crucial that we have \ell_s \rightarrow_{s} 42 in our precondition, since it means we exclusively own the location \ell_s and hence there was no interference by other threads. Specifically, we apply the rule \text{source-load}.

We are left with \{ \ell_t \rightarrow_{t} 42 * \ell_s \rightarrow_{s} 42 \} \leq \{ v_t, v_s, v_t = v_s \}. Using \text{sim-value}, establishing the postcondition is trivial since we only have to show the equality of 42 and 42.

2.2 Interfacing with External Code

The previous optimization exploits ownership to ensure that a memory location cannot be modified between allocation and a load. However, ownership reasoning carries a lot further: we can also use it to argue that external, unknown code in the current thread cannot modify the location.

Function calls. Consider the following example where function f (which we assume to know nothing about) is called between an allocation and a load:

\[
\{ \text{True} \} \text{ let } y := \text{ref}(42) \text{ in } \text{call } f 23; 42 \leq \{ \text{True} \} \text{ let } y := \text{ref}(42) \text{ in } \text{call } f 23; 1y \text{ } v_t, v_s, v_t = v_s \}
\]
The start of the proof proceeds as before: we allocate the locations in source and target, so the program variable \( y \) gets replaced by some location \( \ell_s \) in the source and (potentially different) location \( \ell_t \) in the target. Next, we focus on the call to \( f \) with the sequencing rule \texttt{SIM-BIND}, which enables us to first show a simulation of two subexpressions in evaluation position before considering the surrounding context.

Thus, we are left with \( \{ \ell_t \mapsto^\text{src} 42 * \ell_s \mapsto^\text{arc} 42 \} \) \( \text{call} \ f \ 23 \leq \text{call} \ f \ 23 \ \{ \ell_t \mapsto^\text{src} 42 * \ell_s \mapsto^\text{arc} 42 \} \) (the return value does not matter in this case). We now exploit that our simulation relation allows us to skip over calls to the \textit{same} function \( f \) in source and target, provided that we call it with “sufficiently similar” arguments \( v_s \) and \( v_t \). We get to assume that the function behaves the same on both sides and returns “sufficiently similar” values. Formally, this is captured by \texttt{SIM-CALL}, where the value relation \( \mathcal{V} v_t v_s \) captures the notion of being “similar”. As shown in Figure 4, for simple cases like integers, the value relation just requires syntactic equality, while for composite values like pairs it is lifted componentwise. The more complicated case of locations will be discussed below.

Like all our proof rules, \texttt{SIM-CALL} is compatible with framing, so we can just frame ownership of \( \ell_t \) and \( \ell_s \) around the call. In this case, however, that is a remarkably powerful reasoning principle! It reflects the idea that local locations are not accessible to unknown function calls, and thus not affected by those calls. Separation logic lets us express this at a very high level of abstraction, making this kind of reasoning effortless even when doing mechanized interactive proofs in Coq.

After using \texttt{SIM-CALL} and \texttt{SIM-FRAME}, we are left with the goal \( \{ \ell_t \mapsto^\text{src} 42 * \ell_s \mapsto^\text{arc} 42 \} \) \( 42 \leq !\ell_s \{ v_t, v_s, v_t = v_s \} \). We can now complete the proof in the same way as before.

But let us backtrack for a moment: how can it be sound to just skip over function calls for arbitrary functions \( f \)? Intuitively, a compiler should be able to reason locally about function bodies, without making strong assumptions about the behavior of unknown functions the program might be linked against. The key to enabling this is to make our simulation relation \textit{open}, allowing to skip calls to the \textit{same} function in the source and target. But of course, we have to work for this: to enable \texttt{SIM-CALL}, the function we skip over needs to respect ownership (which enabled us to frame local ownership around the call)! The top-level soundness proof (§5) will thus assume that \textit{all} functions in the program satisfy the simulation relation, tying a big mutually recursive knot.

### Passing pointers to functions.

Above, we have postponed discussion of the value relation for locations. To illustrate what happens if we pass a location to a function, we consider the following example: we optimize away a load from a pointer that we pass to a function afterwards.

\[
\begin{align*}
&\{ \text{True} \} \quad \text{let} \ y := \text{ref}(42) \ \text{in} \quad \text{let} \ z := \text{ref}(42) \ \text{in} \\
&\quad \text{call} \ f \ y; !y + z \quad \leq \quad \text{call} \ f \ y; !y + 42 \quad \{ v_t, v_s, v_t = v_s \}
\end{align*}
\]

As before, we start by (1) allocating the pointers in source and target and (2) loading from the pointer in the source, leveraging local ownership. We end up with the goal:

\[
\{ \ell_t \mapsto^\text{src} 42 * \ell_s \mapsto^\text{arc} 42 \} \ \text{call} \ f \ y; !y + 42 \leq \text{call} \ f \ y; !y + 42 \ \{ v_t, v_s, v_t = v_s \}
\]
When passing pointers to unknown functions, we have to ensure that they in turn point to “sufficiently similar” values, as those can be observed by the callee. Moreover, the location must not be exclusively owned, since the callee could mutate memory and thus violate the ownership discipline. Both of these aspects are captured by the assertion $\ell_t \leftrightarrow_h \ell_s$ which states that target location $\ell_t$ and source location $\ell_s$ form an escaped pair of locations. We can convert locally owned locations to escaped locations by giving up ownership with the rule LOC-ESCAPE:

$$\ell_t \leftrightarrow \text{loc-esc} V v \ell_s \leadsto \text{arc} V v \ell_s + P$$

$$\ell_t \leq e_s \{\Phi\}$$

$$\ell_t \leq \text{sim-load-esc} \{\Phi\}$$

The assertion $\ell_t \leftrightarrow_h \ell_s$ represents knowledge, not exclusive ownership, and as such it is freely duplicable. We can thus retain it beyond the function call and are left with the remaining goal $\{\ell_t \leftrightarrow_h \ell_s\} \not\rightarrow [\ell_t + 42 \leq \ell_t + 42 \{v_1, v_2, v_3 = v_4\}]$. Now, for the last step, we load both locations simultaneously in with SIM-LOAD-ESCAPED (see above). Since the loaded values are integers related by $V$, they must therefore be equal, establishing the postcondition. Note that, as we have given up ownership of $\ell_s$ to escape it, we cannot be sure that $\ell_s$ still stores 42, and thus cannot optimize away the load after the call as we did before. This is for a good reason: the function we have passed the location to could just have written a different value to it.

### 2.3 Exploiting Undefined Behavior

So far, we have seen how our simulation relation allows us to prove optimizations that rely on ownership reasoning. We now introduce another core component of Simuliris: the ability to exploit undefined behavior (UB) in the source program. We will later leverage this feature for reasoning about data races (§3), but for now we focus on a simple example:

```plaintext
let (x, y) := call f () in let (x, y) := call f () in
{True} let z := x/y in \leq \{v_f, v_s, True\}
call g z if y \neq 0 then call g z else call h z
```

Here, we optimize away the conditional and, instead, always pick the "then" branch in the optimized program. The reason why this optimization is correct is that, in Simulang (similar to C), division by 0 is UB. As a compiler, we may assume that the original program (the source program) does not have UB. It then follows that in all relevant (UB-free) executions, $y \neq 0$.

In Simuliris, we have additional reasoning principles to leverage the assumption that the source has no UB: the judgment $e_s \leadsto Q$ expresses that $Q$ follows from the assumption that $e_s$ does not have UB. The relevant rules for our example are given in Figure 5. In the case of division, we can use them to prove the quadruple {True} $v/w \leq v/w \{v_f, v_s, v_i = v_5 \ast w \neq 0\}$. This quadruple not only ensures that the resulting values are the same, but also that the value $w$, the divisor, is not 0. Importantly, the fact that $w$ is not 0 is not a precondition that we need to provide—it appears in our postcondition, so it is something we learn! We can then use $w \neq 0$ afterwards to prove that the conditional will always pick the left branch.

### 2.4 Coinductive Reasoning

All the examples we have considered so far did not contain any loops. We now show that the above reasoning principles extend naturally to programs with loops. To this end, we use the example of hoisting a load out of a loop. More precisely, as depicted in Figure 6, we avoid repeated loads from the pointer $x$ inside a while-loop and, instead, load only once from $x$ before the loop starts. The

---

6Here we implicitly exploit that addition would go wrong if these were not integers; this technique is described more precisely in the next subsection.
condition and body of the loop are defined by arbitrary functions \( f \) and \( g \), demonstrating that the reasoning works without any knowledge of what the loop is actually doing or whether it even terminates. (One could trivially optimize this code further, but we focus on loop reasoning here.)

Intuitively, this optimization is correct because we have local ownership of \( x \) and, additionally, \( x \) is not modified in the loop. We will now make this argument formal in Simuliris. The initial allocation of \( x \) and also the new load in the target are covered using the principles we have seen before. What is more interesting is what we do about the loop: we will use a coinductive argument to justify this optimization. Specifically, we will use the following coinduction principle for while-loops:

\[
\begin{align*}
\text{while-coind} & \\
\{I\} c_t \leq c_s \{v_t, v_s. \forall V. v_t \vee v_s \ast ((v_s = \text{true} \ast I) \vee (v_s \neq \text{true} \ast Q)) \} & \Rightarrow \{I\} e_t \leq e_s \{I\}
\end{align*}
\]

The idea of this rule is the following: when we apply it, we may pick a loop invariant \( I \). We then first show that given the invariant \( I \), either the loop conditions \( c_t \) and \( c_s \) (ordinary expressions) both evaluate to \text{true} and the invariant still holds, or they both do not evaluate to \text{true} and the postcondition \( Q \) of the loop holds. (The postcondition is not simply the negation of the loop condition, because we also need to transfer ownership.) Second, we need to show that if we execute the loop bodies \( e_t \) and \( e_s \), then the loop invariant \( I \) is preserved.

Applying this rule is the key step of verifying the example in Figure 6. At the point where we have to reason about the loop, our current precondition is \( \ell_t \leftarrow^{\text{init}} 0 \ast \ell_s \leftarrow^{\text{init}} 0 \) where \( \ell_t \) and \( \ell_s \) are the source and target values of \( x \). We proceed by applying first \text{sim-bind} to focus on the loops and then \text{while-coind} to reason about the loop. As the invariant, we pick \( I \triangleq (\ell_t \leftarrow^{\text{init}} 0 \ast \ell_s \leftarrow^{\text{init}} 0) \) (our precondition). It is straightforward to show both premises of the rule \text{while-coind}.

**Parameterized coinduction.** The rule \text{while-coind} is already sufficient to verify the previous example. However, there are interesting loops for which \text{while-coind} is not strong enough. For example, what if our loop invariant \( I \) only holds after every other iteration (e.g., it asserts that some loop counter is even)? To also cover such cases, our simulation supports parameterized coinduction [Hur et al. 2013]. Whereas regular coinduction requires re-establishing the invariant after each iteration, parameterized coinduction says that we can take any number of rounds through the loop and then re-establish the invariant whenever we are “back at the beginning”:

\[
\begin{align*}
\text{while-paco} & \\
W_t & = \text{while } c_t \text{ do } e_t \text{ od} \quad W_s = \text{while } c_s \text{ do } e_s \text{ od} \\
\{I\} \text{ if } c_t \text{ then } e_t; W_t \text{ else } () & \leq \{I\} \text{ if } c_s \text{ then } e_s; W_s \text{ else } () \{e'_t, e'_s. \Phi e'_t e'_s \vee (e'_t = W_t \ast e'_s = W_s \ast I)\} \\
\{I\} W_t & \leq W_s \{e'_t, e'_s. \Phi e'_t e'_s\}
\end{align*}
\]
In **while-paco**, the while loops are first reduced for one step, entering the first loop iteration (as a guard for the coinduction). Thereafter, we can finish the proof—at any point—by showing either (1) the postcondition \( \Phi \) or (2) that the current target and source expressions are the \( W_t \) and \( W_s \) that we started with and the invariant \( I \) holds again. The latter condition is expressed using a **postcondition on expressions**. That is, we generalize our simulation relation (and the source/target triples) from a postcondition on values to a postcondition on expressions, such that one can finish the proof at any time if the current expressions satisfy the postcondition: \( \{ \Phi \ e_t \ e_s \} \ e_t \leq e_s \ \{ e_t' \ e_s' \ \Phi \ e_t' \ e_s' \} \). The reasoning principles that we have seen so far (e.g., **sim-bind**, **source-focus**, and **sim-frame**) all generalize to expression postconditions. An overview of the more general rules with expression postconditions can be found in the supplementary material [Anonymous 2021].

3 **SIMULIRIS BY EXAMPLE: EXPLOITING NON-ATOMIC ACCESSES**

Now that we have seen the basics of Simuliris, let us look at a class of challenging optimizations where the ownership reasoning provided by Simuliris enables an elegant verification technique: optimizations around non-atomic accesses as found in languages like C. First, we introduce the basic principles of non-atomic accesses (§3.1), then we introduce proof rules for verifying optimizations around such accesses (§3.2) and finally we verify the optimization from the introduction (§3.3).

3.1 **Basics of Non-atomics**

Previously, we have seen how ownership enables a wide variety of optimizations (§2) for local memory locations that have not been leaked to other, unknown code yet. However, what about escaped locations? At first, it may seem impossible to optimize accesses to escaped locations since other threads might interfere and invalidate the optimization by overwriting the memory in a racy way. This is reflected in the fact that one has to give up ownership of the location when escaping it.

However, compilers have another trick up their sleeve to enable optimizations even for escaped locations: it is based on the observation that only very few accesses to escaped locations actually race with other threads (which would invalidate optimizations), so the programmer should mark such accesses as **atomic** accesses. Atomic accesses can be used to provide synchronization between different threads and are for example used for the implementation of synchronization primitives like locks. For all other accesses, called **non-atomic** accesses, the programmer guarantees that they will not race with other threads and thus they can be optimized more heavily. Technically, this works by assigning **undefined behavior** (UB) to executions with races on non-atomic accesses and thus such executions do not have to be considered when verifying an optimization.

In this paper, we follow the definition of non-atomic accesses used by RustBelt [Jung et al. 2018a] and RefinedC [Sammler et al. 2021]: first, we distinguish between non-atomic accesses (\( x \leftarrow y \) and \( !x \)) and sequentially consistent atomic accesses (\( x \leftarrow^{sc} y \) and \( !^{sc}x \)). Second, we extend the operational semantics with a data-race detector that raises UB if two threads access the same location at the same time, where at least one access is a store and at least one access is non-atomic.\(^7\)

As an example of optimizations enabled by these semantics, consider the following transformation that replaces a load from \( x \) with the value of a previous (non-atomic) store (assuming that the omitted code in ... does not write to \( x \) or perform atomic stores):

\[
\begin{align*}
x & \leftarrow 42; \ldots ; \!x \xrightarrow{\text{optimized}} x \leftarrow 42; \ldots ; 42
\end{align*}
\]

The optimization can be justified as follows: if there is no store to \( x \) by another thread between the first and the second statement, \( x \) still contains the value 42 (because the code in ... does not write to \( x \) and the optimization is correct. Otherwise, the programs behave differently, but the optimization

\(^7\)A detailed description of the data-race detector can be found in [Jung 2020, §9.2] and [Anonymous 2021, §2].
We have seen that the correctness of optimizations of non-atomic accesses relies on some subtle which in turn enables many of the proof rules seen before for local locations to apply. For the verification technique that abstracts over these details and only exposes simple yet powerful, Vol. 1, No. 1, Article . Publication date: October 2021.

is still correct because we can show that this case is impossible for well-defined executions (without UB). We do this as follows: we know that there is another thread that performs a store to x. If this concurrent store occurs at the same time as our store to x, the race detector raises UB and we are done. However, what if in the execution we are considering, the other store is delayed, so that the two stores do not occur at the same time? The race detector would miss that data race. But in this case we can construct an alternative thread interleaving where the concurrent stores do happen at the same time, and in this interleaving the race detector raises UB—this reordering is possible because there is no atomic store in the omitted code. Since we can assume that no interleaving raises UB, we are done.

3.2 Justifying Optimizations of Non-atomic Accesses via Ownership

We have seen that the correctness of optimizations of non-atomic accesses relies on some subtle reasoning about UB and alternative thread interleavings. This section shows how we build a verification technique that abstracts over these details and only exposes simple yet powerful separation logic rules for justifying optimizations around non-atomic accesses. The key insight is that we can use the presence of a non-atomic accesses to gain ownership of exposed locations, which in turn enables many of the proof rules seen before for local locations to apply. For the example above, we know that no other thread could access the location x without raising UB, giving us exclusive ownership of x. With this ownership, we can prove that the load of x returns 42. To make this argument formal, we prove the following quadruple:

\[ \{ x_t \leftrightarrow_h \rightarrow x_s \rightarrow y_r \rightarrow y_s \rightarrow \text{exploit } \pi \} \]

\[ x_t \leftarrow 42; !^c y_r ; 42 \leq \pi \rightarrow x_\pi \rightarrow 42; !^c y_s ; !x_s \]

\[ \{ v_t, v_s, v_\pi = v_s \rightarrow \text{exploit } \pi \} \]

Note that this quadruple uses an atomic load from an escaped location y as a concrete but interesting instance of the code between the store and the load.\(^5\) We also need a new logical assertion, exploit \(\pi \emptyset\), that we will explain during the proof.

\(^5\)We have also verified this example for arbitrary read-only code between the store and the load, but this requires some further machinery that we omit for the purpose of presentation.
SIM-LOAD-SC
\{C(ℓ_s) = ⊥ \land ℓ_t \leftrightarrow_h ℓ_s \land \text{exploit } \pi \ C \} \text{i} \text{sc } ℓ_t \leq_π \text{i} \text{sc } ℓ_s \{v_t, v_s. \{V \ y_t v_s \land \text{exploit } \pi \ C \}

SIM-LOAD-NA
\{C(ℓ_s) = ⊥ \land ℓ_t \leftrightarrow_h ℓ_s \land \text{exploit } \pi \ C \} \text{i} \text{π } ℓ_t \leq_π \text{i} \text{π } ℓ_s \{v_t, v_s. \{V \ y_t v_s \land \text{exploit } \pi \ C \}

SIM-STORE-SC
\{ℓ_t \leftrightarrow_h ℓ_s \land ℓ_s \land \text{exploit } \pi \ 0 \} ℓ_t \leftarrow \text{sc } v_t \leq_π ℓ_s \leftarrow \text{sc } v_s \{v'_t, v'_s. v'_t = () \land v'_s = () \land \text{exploit } \pi \ 0 \}

SIM-CALL-REVISED
\{V \ y_t v_s \land \text{exploit } \pi \ 0 \} \text{call } f v_t \leq_π \text{call } f v_s \{v'_t, v'_s. \{V \ y'_t v'_s \land \text{exploit } \pi \ 0 \}

Fig. 8. Some revised rules.

Rules for exploiting UB of non-atomic accesses. The first step of verifying the example is to gain ownership of \(x_4\) and \(x_5\) by “exploiting” the non-atomic store to \(x_4\). We do this via the rule EXPLOIT-STORE in Figure 7: it lets us acquire ownership of a source location \(ℓ_s\) (here \(x_4 \mapsto_{\text{arc}} v_5\)) and related target location \(ℓ_t\) (here \(x_t \mapsto_{\text{lft}} v_t\)) given a reachable non-atomic store in the source program.\(^9\) These ownership assertions are the same as for local locations, which consequently allows us to use the rules for loads and stores presented in Figure 3 (§2.1) during the verification of our example. However, we need to ensure that EXPLOIT-STORE is not used twice for the same location, as it is not sound to gain exclusive ownership of the same location twice. This is achieved via the assertion exploit \(\pi \ C\), which tracks which locations have been exploited by the current thread \(\pi\) via EXPLOIT-STORE. (exploit \(\pi \ C\) is also relevant for other rules as we will see later.) Concretely, EXPLOIT-STORE first checks that \(ℓ_s\) is currently not already exploited (via the side-condition \(C(ℓ_s) = \bot\) which holds trivially in our example since \(C = \emptyset\)). Then it records that one has acquired ownership for \(ℓ_s\) via a store (denoted by \(W\) for “write”) in \(C\) (resulting in exploit \(\pi \ (x_s \mapsto W)\) in our example). To be able to talk about the “current thread”, we also equip the simulation relation with a thread id \(\pi\).

The rule for exploiting loads EXPLOIT-LOAD is similar to EXPLOIT-STORE except that it does not provide full ownership but only fractional ownership [Boyland 2003; Bornat et al. 2005] with some fraction \(q\). This gives us permission to introduce extra loads in the target program, but we cannot introduce extra stores. Correspondingly, EXPLOIT-LOAD records that one has acquired the fraction \(q\) by adding \(R(q)\) (for “read”) to exploit \(\pi \ C\).

For our example, after using EXPLOIT-STORE and obtaining ownership of \(x_t \mapsto_{\text{lft}} v_t\) and \(x_s \mapsto_{\text{arc}} v_5\) for some \(v_t, v_5\), we execute the stores in source and target and transform the ownership to \(x_t \mapsto_{\text{lft}} 42\) and \(x_s \mapsto_{\text{arc}} 42\). Next, we verify the atomic load in both source and target via the rule SIM-LOAD-SC in Figure 8. This rule is similar to SIM-LOAD-ESCAPED except that we need to prove that the location \(ℓ_s\) (here \(y_4\)) is not in \(C\). (In fact, as we will see in §6, SIM-LOAD-ESCAPED is no longer sound and must be replaced by SIM-LOAD-NA with a similar sidecondition.) We only consider the case \(y_s \neq y_4\), where this sidecondition holds. (For \(y_s = y_4\), we can use the ownership of \(x_3\) gained earlier to perform the loads.) For the final load, we use SOURCE-LOAD to prove that the load in the source returns 42, which ensures that both sides return the same value. Finally, we reestablish exploit \(\pi \ 0\) by giving back \(x_t \mapsto_{\text{lft}} 42\) and \(x_s \mapsto_{\text{arc}} 42\) via RELEASE-EXPLOIT, which lets us end exploiting a location.

Returning ownership and revised rules. At this point, one may wonder what forces us to ever give up ownership of exploited locations. Intuitively, it is only sound to retain ownership until

\(^9\)The notion of reachability used by EXPLOIT-STORE will be explained in §3.3 where it is used in a non-trivial way.
\{x_i \leftrightarrow_h x_s \ast y_t \leftrightarrow_h y_s \ast \text{exploit } \pi \emptyset\}

\begin{align*}
&\text{let } (n, m) := (lx_t, ly_t) \text{ in} \\
&\text{let } (i, r) := (\text{ref}(0), \text{ref}(m)) \text{ in} \\
&\text{while } !i \neq n \text{ do} \\
&\quad i \leftarrow !i + 1; r \leftarrow !r + m \\
&\text{od}; !r
\end{align*}

\{v_r, v_s, v_t = v_s \ast \text{exploit } \pi \emptyset\}

Fig. 9. Simulation for the optimization from §1.

the thread synchronizes with other threads—if there is proper synchronization, there is no data race, and thus no UB. It turns out that for SimuLang, we only need to be concerned with atomic stores here: atomic loads do not release any information to other threads, so they cannot be used by those threads to avoid data races. Hence, the rule for atomic stores \textsc{sim-store-sc} in Figure 8 requires giving up ownership of all exploited locations.\(^{10}\) This is enforced via the precondition exploit \(\pi \emptyset\): to establish exploit \(\pi \emptyset\), one has to stop exploiting any location by applying \textsc{release-exploit}, which in turn requires giving up ownership of the exploited locations. The values stored at these locations must be related by \(V\) to reestablish the invariant that all escaped non-exploited locations contain related values (similar to \textsc{loc-escape}).

The exploit \(\pi \emptyset\) precondition of \textsc{sim-store-sc} also explains why the postcondition in the optimization example needs to contain exploit \(\pi \emptyset\): since we do not know the following code, it could perform an atomic store so we need to provide it with exploit \(\pi \emptyset\). For similar reasons, the revised rule \textsc{sim-call-revised} for calling unknown functions also has that precondition (and postcondition).

3.3 Combining Data-Race Exploitation and Coinductive Reasoning

To finish up the expository part of this paper, we revisit the example from §1. This will demonstrate the benefit of the unifying logic provided by Simuliris: we can seamlessly combine the reasoning principles presented in this section with the coinduction principles for loops from §2.4 to verify the challenging optimization in Figure 9, hoisting read accesses out of a (potentially diverging) loop.

The source program initializes a counter \(i\) and a result accumulator \(r\). Then, it enters a loop where each iteration increments \(r\) by the value stored at \(y_s\) and increments \(i\) by \(1\), until \(i\) reaches the value stored at \(x_s\). The optimized target program replaces the repeated loads of \(x_s\) and \(y_s\) with a single load for each location, using \(n\) and \(m\) to store the loaded values.

With the rules in Figure 8 and \textsc{while-coind} it is straightforward to verify the optimization in Figure 9: first, we use \textsc{exploit-load} twice to gain ownership of \(x_s\), \(x_t\), \(y_s\), and \(y_t\).\(^{11}\) Here, we leverage that it is sufficient for the non-atomic load to be reachable for \textsc{exploit-load} to apply, exploiting that the notion of reachability \(\rightarrow_s^c\) used by \textsc{exploit-load} allows skipping over impure operations (like allocation or memory accesses including atomic loads and stores) and allows assuming that there is no UB (in particular, all loads and stores will succeed).\(^{12}\)

After this, the rest of the verification is standard separation logic reasoning: we begin with the loads from \(x_t\) and \(y_t\) in the target program, then we allocate \(i\) and \(r\) on both sides—using the

\(^{10}\)We have also verified a stronger rule that allows retaining ownership over atomic stores if there is another non-atomic access reachable after the atomic store. This allows to e.g., reorder non-atomic accesses before atomic stores.

\(^{11}\)We only present the case were \(x_s \neq y_s\). The \(x_s = y_s\) case works the same except that we only use \textsc{exploit-load} once.

\(^{12}\)The load has to be reachable from an arbitrary starting state and the resulting state is ignored.
We have now demonstrated how to prove relational specifications, expressed via Hoare quadruples.

To define contextual refinement, we start by defining whole-program refinement. This is basically

\[ (K[\{e\}], \sigma) \leadsto (K[], \sigma, [e]) \]

\[ (e, \sigma) \leadsto (e', \sigma', \bar{e_f}) \]

\[ (T_1 + [e] + T_2, \sigma) \overset{\delta_{tp}}{\to} (T_1 + [e'] + T_2 + \bar{e_f}, \sigma') \]

Fig. 10. Excerpt of the operational semantics of SimuLang.

ownership of \( y_s \) to justify \(!y_s \). After that, we apply \textsc{while-coind} with the loop invariant:

\[ \exists z_i, v'_s, v^y_s, v^x_s, v'_f, v^y_s, v^x_s. i_t \leftarrow \text{last } z_i \ast i_s \leftarrow \text{arc } z_i \ast r_t \leftarrow \text{last } v'^y_f \ast r_s \leftarrow \text{arc } v'^y_s \ast x_t \overset{\delta_{arc}}{\mapsto} v^y_s \ast x_s \overset{\delta_{arc}}{\mapsto} v^x_s \ast \text{exploit } \pi (y_s \mapsto R(q_y), x_s \mapsto R(q_x)) \]

This invariant might seem complicated, but it just contains ownership of all source and target locations and ensures that they point to related values. With this invariant, it is straightforward to verify the rest of the function. The optimization is justified because the loop invariant ensures that \( x_s \) always points to \( v_s \) when the condition executes, which is related by \( V \) to the value used in the target program—this makes sure that the comparison evaluates the same. A similar argument applies for optimizing the non-atomic load in the loop.

4 CONTEXTUAL REFINEMENT

We have now demonstrated how to prove relational specifications, expressed via Hoare quadruples. However, what we actually want to prove is that an optimizing compiler may replace the code on the right-hand side of such a simulation by the code on the left-hand side. This is called contextual refinement. For the simple data race example, the statement would look something like this:

\[ x \leftarrow 42; !^e y; 42 \equiv_{\text{ctx}} x \leftarrow 42; !^e y; !x \]

Note that \( x \) and \( y \) are free (program) variables in these code snippets, unlike previously where we already replaced them by logic-level variables \( x_s, x_t, y_s, \) and \( y_f \).

In the following, we will first explain how contextual refinement is defined for SimuLang, and then we will show how it can be proven.

4.1 Fair Termination-Preserving Contextual Refinement

To define contextual refinement, we start by defining whole-program refinement. This is basically standard, except that we have to account for our particular notion of “whole programs”.

As already shown in Figure 2, a whole program is given by a list of function declarations. In Figure 10, we show an excerpt of how program execution is defined: the current machine configuration is given by a list of expressions (the thread pool) and the global state. The thread-pool step relation \( \to_{tp} \) picks an arbitrary thread and performs a per-thread reduction step \( \to \) . (Here, \( \oplus \) is list concatenation.) As an example of a per-thread step, we show the reduction rule for \textsc{fork}: the last component of the per-thread step relation indicates a list of new threads that are created by this step. All step relations are indexed by the list of functions \( p \), which is relevant for \textsc{call} expressions.

To execute a given program \( p \), we assume a well-known function name main indicating the entry point of the program, so that the initial machine configuration of a whole-program execution is given by \( I \triangleq ([\text{call } \text{main } ()], \emptyset) \): a single main thread, and the empty heap.\(^{13}\) The set of possible

\(^{13}\)In our formal development, we also support non-empty initial heaps.
behaviors of this program is then defined by

\[ B(\rho) \triangleq \left\{ V(\nu) \mid I \text{ has a finite } \mathcal{L}_{\text{tp}} \text{ execution where the main thread returns } \nu \right\} \]

∪ \left\{ \omega \mid I \text{ has a fair, infinite } \mathcal{L}_{\text{tp}} \text{ execution} \right\}

∪ \left\{ T \mid I \text{ can via } \mathcal{L}_{\text{tp}} \text{ reach a configuration which is stuck} \right\}

We define a notion of refinement on these behaviors as follows:

\[ V(\nu_i) \sqsubseteq_{\text{beh}} V(\nu_s) \text{ if } O(\nu_i, \nu_s), \quad \omega \sqsubseteq_{\text{beh}} \omega, \quad b \sqsubseteq_{\text{beh}} T \]

In particular, if a program has undefined behavior in the source, then any target behavior is allowed. Here, \( O \) defines the possible observations on return values. For SimuLang, this is mostly defined as reflexivity, except that we assume that no observations are possible on locations, so \( O(\ell_i, \ell_s) \) holds for any two locations (as is standard). We can lift this notion of refinement to programs:

\[ \rho_t \sqsubseteq_{\text{prog}} \rho_s \triangleq \forall b_t \in B(\rho_t). \exists b_s \in B(\rho_s). b_t \sqsubseteq_{\text{beh}} b_s \]

And finally, we lift this to contextual refinement of open terms by quantifying over an arbitrary closing context:

\[ e_t \sqsubseteq_{\text{ctx}} e_s \triangleq \forall \rho, f, x, C. \left( \text{FreeVars}(C[e_t]) \cup \text{FreeVars}(C[e_s]) \subseteq \{x\} \right) \land \text{wf}(C) \land

\left( V(f' x' \triangleq e') \in \rho. \text{FreeVars}(e') \subseteq \{x'\} \land \text{wf}(e') \right) \Rightarrow

(f x \triangleq C[e_t], \rho) \sqsubseteq_{\text{prog}} (f x \triangleq C[e_s], \rho) \]

The “closing context” consists of two parts: a context \( C \) (i.e., an expression with a hole at an arbitrary place) to turn the expressions into function bodies (with the argument variable \( y \) as the only free variable), and a program \( \rho \) to supply all the other functions of the program (which must also be appropriately closed). Furthermore, both context and program must not contain any location values, which is captured by the well-formedness predicate \( \text{wf} \).

4.2 Logical Relation

Now that we have defined contextual refinement, the question is—how can we prove it? Here we follow the usual recipe of logical relations. We already have a powerful simulation relation that works on closed expressions. We can lift it to open expressions by quantifying over a closing substitution \( \gamma \) that replaces free variables by related values—here we reuse the value relation \( \text{Val} \) that we already used for external function calls:

\[ e_t \leq_{\text{log}} e_s \triangleq \forall \gamma : \text{FreeVars}(e_t) \cup \text{FreeVars}(e_s) \rightarrow \text{Val} \times \text{Val}.

\{ \text{exploit } \pi \emptyset \ast \forall x, v_i, v_s. y(x) = (v_i, v_s) \Rightarrow \forall V \ni v_i, v_s \} \gamma_{\text{tgt}}(e_t) \leq \gamma_{\text{src}}(e_s) \{ v_i', v_s'. \text{exploit } \pi \emptyset \ast V' v_i' v_s' \}

Here we use \( \gamma_{\text{src}} \) and \( \gamma_{\text{tgt}} \) to refer to the source and target projections of \( \gamma \) (which assigns two values to each free variable), respectively. The final values that the terms reduce to must also be in the value relation. As in SIM-CALL-REVISED, we also ensure no data-race exploitation is ongoing at the beginning and end of this simulation.

As one would expect from a logical relation, \( \leq_{\text{log}} \) is compatible with all expression formers, in the sense that when all subexpressions are related, then so are the compound expressions. This gives rise to the fundamental theorem of our logical relation, and the fact that it is a precongruence with respect to language contexts:

**Theorem 4.1 (Fundamental Theorem).** Let \( e \) be a well-formed expression. Then \( e \leq_{\text{log}} e \).

**Theorem 4.2 (Contextual closure).** Assume \( e_t \leq_{\text{log}} e_s \) and \( \text{wf}(C) \). Then \( C[e_t] \leq_{\text{log}} C[e_s] \).
As usual, the fundamental theorem establishes that the logical relation is reflexive—for terms that do not contain literal location values ℓ.

How does this help us to establish our desired contextual refinement? If we plug the two terms from that example into our logical relation, we obtain almost exactly the statement we proved in §3.2. (Technically, we only get that the free variables are in the value relation, not that they are related escaped locations—but the proof is easily adjusted to this.) We thus have already proven that our optimization is in the logical relation. Now we are just one theorem away from our goal of contextual refinement between these two terms:

**Theorem 4.3 (Adequacy).** If $e_t \leq_{\text{log}} e_s$, then $e_t \subseteq_{\text{ctx}} e_s$.

In other words, to prove correctness of optimizations that replace $e_s$ by $e_t$ anywhere in the program, it is sufficient to establish $e_t \leq_{\text{log}} e_s$—which we can in turn do with the proof techniques we have introduced in the previous two sections.

## 5 Model and Adequacy

Now that we have stated adequacy, the question of course is: how can we prove it? To discuss this proof, we first explain how our relational Hoare quadruples are defined (in §5.1), and then we give an idea of the key lemma that enables the adequacy proof of the logical relation (in §5.2). Most of what we discuss in this section is defined in a language-generic way, so while the definitions and proofs are involved, they have to be done only once and can be reused many times.

### 5.1 Simulation Relation

The simulation relation $\{P\} e_t \leq e_s \{\Phi\}$ forms the heart of Simuliris. It is itself defined in separation logic; more specifically, it is defined in a variant of Iris [Jung et al. 2018b] that we dub Iris$^\text{light}$. Iris$^\text{light}$ is essentially “Iris without the step-indexing”: we keep the basic logic of bunched implications [O’Hearn and Pym 1999] and the persistence modality, but we ignore all the step-indexing aspects of Iris (e.g., the later modality $\triangleright P$, guarded recursion, and impredicative invariants). Technically, instead of defining a new logic, we found it easier to literally reuse the model of Iris itself, but fix the step-index to be 0. This lets us directly use all of the Iris infrastructure.

Inspired by Iris’s weakest precondition, we define the simulation relation as

$$\{P\} e_t \leq e_s \{\Phi\} \triangleq (P \leftarrow \text{sim} e_t e_s \Phi)$$

where $\text{sim} e_t e_s \Phi$ is the weakest precondition that we need to impose such that $e_s$ simulates $e_t$ with postcondition $\Phi$, and the persistence modality $\Box$ ensures that the simulation relation is duplicable.

The interesting part of this definition is, of course, $\text{sim} e_t e_s \Phi$ itself. A simplified version of $\text{sim} e_t e_s \Phi$ is shown in Figure 11, with a focus on our four essential features: ownership reasoning, undefined behavior, concurrency, and fair termination preservation.

We assume that the setup of the language roughly matches SimuLang in the sense that there is a per-thread step relation $\rightarrow^\text{light}$ indexed by some “program” that maps function names to “function bodies” (the definition of which is left up to the language). We write $(e, \sigma) \rightarrow^\text{light} (e', \sigma')$ for a step where no threads are forked off. Finally, we say that an expression $e$ is safe in state $\sigma$ and program $\rho$, written safe$(\rho, e, \sigma)$, if it cannot reach a stuck configuration, i.e., a state and expression which is neither a value nor reducible.

We start the explanation of $\text{sim}$ with a discussion of the general structure, before we focus on the four essential features. In general, the simulation weakest precondition factors into two cases:

---

14The simplified version glosses over two details: an extension for exploiting UB of data races (see §6) and an additional case for skipping function calls to obtain an open simulation. A complete definition can be found in the supplementary material [Anonymous 2021, §1].
\[
\text{sim } e_t \ e_s \ \Phi = \forall \sigma_t, \sigma_s. \ S(\sigma_t, \sigma_s) * \text{safe}(\rho_s, e_s, \sigma_s) \rightarrow \mathcal{R} \\
\begin{cases}
\text{Base case: } \exists e'_s, \sigma'_s. (e_s, \sigma_s) \xrightarrow{\mathcal{L}} (e'_s, \sigma'_s) * S(\sigma'_s, \sigma'_s) * \Phi e_t e'_s \lor \\
\text{Step case: } \text{reducible}(\rho_t, e_t, \sigma_t) * \forall e'_s, \sigma'_s, T_s. (e_s, \sigma_s) \xrightarrow{\mathcal{L}} (e'_s, \sigma'_s, T_s) \rightarrow \mathcal{R} \\
& (\text{Source stutter: } T_s = [] * S(\sigma'_s, \sigma'_s) * \text{sim } e'_t e_s \ \Phi) \lor \\
& (\text{Source step: } \exists e'_s, e''_s, \sigma'_s, \sigma''_s, T_s. (e_s, \sigma_s) \xrightarrow{\mathcal{L}} (e'_s, \sigma'_s) * (e''_s, \sigma''_s, T_s) * S(\sigma''_s, \sigma''_s) * \text{sim } e'_t e''_s \ \Phi * |T_s| = |T_s| * (e_t, e_s) \in \text{zip}(T_t, T_s))
\end{cases}
\]

Fig. 11. Simplified simulation weakest precondition.

the base case and the step case. In the base case, we have reached the postcondition: we allow executing some more steps of the source expression (for target stuttering) and afterwards we prove the postcondition \( \Phi e_t e'_t \). (For the common case of value postconditions, this case is only applicable when the target expression is a value.) In the step case, we have to prove that \( e_t \) is reducible and that for each target step we can find a matching source step—or perform source stuttering. In the source stutter case, we can relate the new target expression \( e'_t \) to the same source expression. In the source step case, we have to execute at least one source step (but we can execute more, which again would be target stuttering). The source has to fork exactly as many threads as the target step. We then have to show that the resulting target expression simulates the source expression.

Ownership reasoning. The support for ownership reasoning in Simuliris is inherited from Irislight. In particular, the update modality \( \mathcal{R} P \) expresses that we are allowed to perform frame-preserving updates on the ghost state in order to obtain the resources satisfying \( P \). This incorporates the Iris approach to ghost state [Jung et al. 2018b] into Simuliris.

The state interpretation \( S(\sigma_t, \sigma_s) \) connects that ghost state to the physical states \( \sigma_t \) and \( \sigma_s \) of the target and source program. (This follows the same setup as typical Hoare triples in Iris.) For example, in SimulLang, the state interpretation governs which locations are currently local and which are escaped. It also ensures that all escaped locations store values related in the value relation \( \mathcal{V} \).

Concurrency. Our simulation relation enforces a one-to-one mapping between threads in the source and target program: for each thread forked in the target, there must be a matching thread simulating it in the source. This is restrictive, but sufficient for the vast majority of compiler optimizations—and it is crucial to our proof of fair termination preservation.

Note how in the source step case, we use a separating conjunction to concisely express that the simulation proofs of all the forked-off threads, and of the original thread, are all based on disjoint resources. This ensures that the simulation of one thread does not interfere with the resources required by another thread.

Undefined behavior. We model UB via stuck expressions. Thus, the assumption that the source does not have UB is reflected in \( \text{sim} \) by assuming \( \text{safe}(\rho_s, e_s, \sigma_s) \). This is exploited by the \( e_s \rightsquigarrow Q \) judgement that we have seen in §2.3, which is defined as:

\[
e_s \rightsquigarrow Q \triangleq (\forall \rho_s, \sigma_s. \text{safe}(\rho_s, e_s, \sigma_s) \Rightarrow Q)
\]

Fair termination preservation. Where does fairness and termination preservation factor into the definition? As already mentioned, we simulate threads pairwise. Thus, if we did a lockstep simulation, then fair termination preservation would be easy: for every target step, there would be
exactly one source step of the corresponding thread. Hence, if the target diverges, then the source diverges—and both executions execute threads in the same order, so fairness is preserved.

What makes fairness and termination preservation challenging is implicit stuttering. We have seen that implicit stuttering is quite useful for reasoning about target and source individually without counting the steps (e.g., see source-focus). Here we have to pay the price: given a fair diverging execution of the target, we have to ensure that no source thread gets “forgotten” for too long such that the corresponding source execution is still fair.

To achieve this, we use a mixed fixed-point to define sim. More specifically, we nest two fixed-points: a greatest fixed-point and a least fixed-point. We define \( \text{sim} \triangleq vG.\mu L. \text{simbody}(G, L) \) where simbody is what we obtain in Figure 11 if we replace the red occurrences of sim with \( G \) (“greatest fixed-point”) and the blue occurrences with \( L \) (“least fixed-point”). The resulting definition satisfies \( \text{sim} = \text{simbody}(\text{sim}, \text{sim}) \), but it is neither the greatest nor the least fixed-point—it is in between.

The greatest fixed-point part of it gives rise to a language-independent parametric coinduction principle, from which language-specific rules such as while-paco can be derived. The least fixed-point on the inside ensures that we can stutter the source, but we can only do so finitely often, which will be crucial for the adequacy proof of fair termination preservation.

### 5.2 Adequacy

Equipped with the simulation relation, we can now turn to adequacy of the logical relation (Theorem 4.3). The proof factors into two steps: we first define a logical whole-program relation \( \rho_t \leq_{\text{prog}} \rho_s \) and show that it implies whole-program refinement (this part is language-independent), and then we use that result to show that \( \leq_{\text{log}} \) implies \( \leq_{\text{ctx}} \) (which is language-specific, but fairly straightforward).

We define \( \rho_t \leq_{\text{prog}} \rho_s \) by lifting the simulation relation to whole programs:

\[
\rho_t \leq_{\text{prog}} \rho_s \triangleq \forall (f x \triangleq e_s) \in \rho_s. \exists (f x \triangleq e_t) \in \rho_t. \forall \nu_t, \nu_s. \forall \nu_t, \nu_s. \{V \nu_t \nu_s\} e_t[\nu_t/x] \leq e_s[\nu_s/x] \{V\}
\]

That is, for every source function \( f x \triangleq e_s \), there must be an implementation \( f x \triangleq e_t \) in the target such that if we insert related values into the function bodies \( e_t \) and \( e_s \), then the resulting target expression simulates the resulting source expression. (The definition of how to substitute the argument into the function body is left to the language.) This definition satisfies:

**Lemma 5.1 (Whole-program adequacy).** If \( S(0, 0) * \rho_t \leq_{\text{prog}} \rho_s \), then \( \rho_t \leq_{\text{prog}} \rho_s \).

This lemma is the core of the adequacy proof of Simuiris. It factors into two parts. The first part is concerned with the fact that sim is an open simulation that can skip matching function calls (sim-call), which we omitted from Figure 11. We prove that this open simulation implies a closed simulation that looks almost exactly like Figure 11. This proof ties the big recursive knot over all the mutually recursive function definitions in the source and target programs. It essentially “inlines” the simulation of the calls that were skipped. Since open simulations are a well-known technique [Kang et al. 2015; Hur et al. 2012], we do not focus on them here.

In the second step, we prove that the closed simulation implies whole-program refinement:

\[
\rho_t \leq_{\text{prog}} \rho_s \triangleq \forall b_t \in B(\rho_t). \exists b_s \in B(\rho_s). b_t \equiv_{\text{beh}} b_s.
\]

Roughly speaking, this proof factors into three cases:

1. To prove that the target does not have UB, we leverage reducibility in the step case in Figure 11. Since all target expressions that we reach are reducible, we are never stuck.
2. To prove that the target only terminates with values that are also possible in the source, we leverage the postcondition: if the target terminates, then the postcondition ensures that the source must also terminate in a value and both satisfy \( V \). The value relation, in turn, implies that the result values are related by \( O \).


(3) Proving that a fair diverging target execution implies a fair diverging source execution is by far the most challenging case. Intuitively, we are termination preserving because we do not allow infinite stuttering (since stuttering uses a least fixpoint), and we are fair termination preserving because we simulate threads in a one-to-one correspondence. Unfortunately, there is a caveat to this intuitive argument—in a sense, through stuttering, the order in which threads are executed can change. For example, we can delay the execution of some source steps in one thread until steps in another thread have passed. Nevertheless, the heart of the intuitive argument can still be recovered (with a coinduction and three nested inductions) and, hence, we obtain fair termination preservation.

We now have all puzzle pieces together to prove our main theorem:

**Theorem 5.2.** If $e_t \leq_{\log} e_s$, then $e_t \sqsubseteq_{\text{ctx}} e_s$.

**Proof Sketch.** We assume some well-formed closing program $P$ and a closing context $C$. What remains to show is $(f \times \triangleq C[e_s]), \rho_t \leq_{\text{prog}} (f \times \triangleq C[e_t]), \rho_s$. With Lemma 5.1, it suffices to show $S(\emptyset, \emptyset) \ast (f \times \triangleq C[e_t]), \rho_t \leq_{\text{prog}} (f \times \triangleq C[e_s]), \rho_s$. A proof of the initial state interpretation $S(\emptyset, \emptyset)$ is assumed to be supplied by the specific language instance of Simuliris. The required whole-program refinement follows from $e_t \leq_{\log} e_s$, and contextual closure of $\leq_{\log}$ (for the function $f$), and from reflexivity of the logical relation (for all the other functions).

## 6 SOUNDNESS OF EXPLOITING NON-ATOMIC ACCESSES

This section explains the core of the soundness argument for the rules that exploit undefined behavior of non-atomic accesses for optimizations from §3.

**The state interpretation.** To understand how EXPOSE-BIJECT HANDS out ownership of escaped locations, we first need to see how the ownership of escaped locations is managed in general. The basic idea is that we maintain a bijection between the escaped locations in the source and target programs, and ensure that matching locations carry related values. Formally, this is expressed by the heapbij($P_h$) predicate that is part of the state interpretation $S$ and is (roughly) defined as follows:

$$\text{heapbij}(P_h) \triangleq \exists L. \text{bijauth}(L) \ast \ast (\ell_t, \ell_s) \in L \exists q_h, v_t, v_s, P_h(\ell_s, q_h) \ast \forall v_t \vdash_{\text{ctx}} \ell_t \Rightarrow_{\text{log}} v_t \ast \ell_s \Rightarrow_{\text{log}} v_s$$

Here, the set $L$ is the bijection containing the escaped locations. The assertion bijauth($L$) represents ownership of Iris ghost state which tracks full knowledge of the current bijection. The $\ell_t \Rightarrow_{\text{log}} \ell_s$ that we have already seen talks about the same ghost state and reflects knowledge that the pair $(\ell_t, \ell_s)$ is in the bijection $L$. This is realized with the standard Iris mechanism of “authoritative ghost state” [Jung et al. 2018b]; the details of this technique do not matter here.

heapbij further says that for every pair of locations in $L$, we have ownership of those locations and their values are in the value relation. Since heapbij is part of the state interpretation $S$, each thread can use this ownership to justify executing a step, but it has to give it back at the end of the instruction and cannot keep the ownership between instructions.

However, to justify EXPOSE-BIJECT and EXPOSE-LOAD, we need to remove (some part of) this ownership from heapbij. This is the purpose of the $P_h$ parameter of heapbij: this relation determines which fraction $q_h$ is stored in heapbij for each escaped location $\ell_s$. A simple state interpretation would just use $P_h(\ell_s, q_h) \triangleq q_h = 1$ and for such a state interpretation one can prove SIM-LOAD-ESCAPED from §2.2. However, for exploiting data races we need a more complicated $P_h$ that lets us store less than the full fraction for exploited locations. In particular, assume

---

15For space reasons, we are omitting the details from the paper.

16Technically, heapbij uses a slightly non-standard notion of points-to predicates that is defined for $q \neq 0$ (as True).
collections of exploited locations for all threads (i.e., the collection of thread π is \(\overline{C}_π\) and is exposed through the assertion exploit \(\pi \overline{C}_π\)). Then, for each location \(\ell_q\), the fraction \(q_h\) stored in heapbij and the fractions handed out through exploitation should sum up to 1:

\[
P_h(\overline{C}) (\ell_q, q_h) ≜ 1 = q_h + \sum_{\pi \in (\ell_q)} (c = R(q)) \cdot q : 1
\]

While the above choice of \(P_h\) allows us to remove ownership from heapbij, this does not come for free. In particular, the ownership inside heapbij was used to justify rules like \texttt{SIM-LOAD-ESCAPED}. How can such a rule still be sound if there is no ownership in heapbij? This is where data races come in. The basic idea is simple: we can only remove ownership from heapbij if we prove that each access expecting this ownership to be in heapbij has a data race and thus need not be considered.

Concretely, consider the case where thread π uses \texttt{EXPLOIT-STORE} to remove the ownership of a location \(\ell_q\) from heapbij. Now another thread \(\pi'\) tries to load \(\ell_q\) but does not find the necessary ownership to justify the load in heapbij. So instead we show that there must be an execution with a data race: we know that thread π can reach a non-atomic store (due to the premise of \texttt{EXPLOIT-STORE}) and thread π’ is ready to perform a load—and these two accesses form a data race, so we are done!

To make this argument formal, we need to make sure that for each element \(c\) of \(\overline{C}_π(\ell_q)\) there is a safe configuration (i.e., a thread pool \(T\) and a state \(σ\)), where thread π can reach a non-atomic store (if \(c = W\)) or a non-atomic load (if \(c = R(q)\)).\(^{17}\) That safe configuration must be the current source configuration, except that thread π can differ. Formally speaking:

\[
\text{exploit}_\text{wf}(\rho_s, σ_s, T_s, \overline{C}) \triangleq \forall π, \ell_q, c, \overline{C}_π(\ell_q) = c ⇒ \exists K, e, v. \text{pool}_\text{safe}(\rho_s, T_s[π \rightarrow e], σ_s) \land e \rightarrow^* K[c = W ? \ell_q ← v : !\ell_q]
\]

One can view \text{exploit}_\text{wf} as keeping track of alternative interleavings where thread π is “paused” while the other source threads step along with the target. Since all interleavings must be data-race free, demonstrating a race in one of these alternative interleavings is sufficient.

In the above scenario where the thread π has exploited \(\ell_q\) with \(\overline{C}_π(\ell_q) = W\) and another thread \(\pi'\) is ready to perform a load of \(\ell_q\) (i.e., \(T_s(\pi') = K'[!\ell_q]\)), \text{exploit}_\text{wf} implies \text{pool}_\text{safe}(\rho_s, T_s[π' \rightarrow K'[!\ell_q]][π \rightarrow K[\ell_q \leftarrow v]], σ_s) (by executing thread π to reach the store). This supposedly safe thread has a data race between the load and the store, and thus is not safe—a contradiction!

To finish the proof of the rule for loading from escaped locations, we need to consider one more case: what if the thread π that exploited \(\ell_q\) is also the thread π’ performing the load, i.e., \(π = π'\)? Then there is no data race as a thread cannot race with itself. Thus, the load rule cannot apply in this situation. We hence weaken \texttt{SIM-LOAD-ESCAPED} to not permit loading from locations that are being exploited by the current thread and arrive at \texttt{SIM-LOAD-NA}. This argument also works for atomic loads and thus also gives us a proof of \texttt{SIM-LOAD-SC}.

Now we have all the ingredients to define the state interpretation \(S\):

\[
S(\ldots, ρ_s, σ_s, T_s) \triangleq \ldots \star \overline{C}. \text{exploitauth}(\overline{C}) \star \text{exploit}_\text{wf}(\rho_s, σ_s, T_s, \overline{C}) \star \text{heapbij}(P_h(\overline{C}))
\]

We make use of the fact that our full simulation relation gives the state interpretation access to the source program \(ρ_s\) and the current source thread pool \(T_s\). We omit the parts of the state interpretation that are responsible for defining the points-to predicates. C is linked to the exploit \(π C\) assertions via the authoritative exploitauth(\(\overline{C}\)). exploit_\text{wf} and heapbij appear as described above.

**Proving \texttt{EXPLOIT-STORE}**. With the state interpretation at hand, let us see how we can prove \texttt{EXPLOIT-STORE} (\texttt{EXPLOIT-LOAD} is analogous), assuming exploit π C and \(\ell_q \leftarrow v_h\) from the precondition as well as \(C(\ell_q) = \bot\) and \(e_s \rightarrow^* K[\ell_q \leftarrow v_0]\). There are two cases to consider.

\(^{17}\)\text{pool}_\text{safe}(\rho_s, T_s, σ_s) is similar to \text{safe}(\rho_s, e_s, σ_s) except for a thread pool \(T_s\) instead of an expression \(e_s\).
Either no thread has exploited \( \ell_\ell \). In this case we know that heapbij contains the full ownership of \( \ell_\ell \) and \( \ell_\ell \). We add \( \ell_\ell \to W \to C \) which allows us to remove this ownership from heapbij and we can establish exploit_wf for the modified collection by adding pool_safe(\( \rho_\ell, T_\ell, \sigma_\ell \)) for the current thread pool and source state.\(^{18}\)

Alternatively, there is another thread that has exploited \( \ell_\ell \) (we know that this cannot be the current thread thanks to the \( C(\ell_\ell) = 1 \) precondition). But then we can construct a data race between the non-atomic access used to justify the exploitation by the other thread and the non-atomic access given to \textbf{EXPLOIT-STORE} and we are done.

**Maintaining the state interpretation.** We have seen proof sketches for some of the rules, but there is an important proof obligation that we have not discussed so far: for each operation we need to prove that it maintains the state interpretation. Since the state interpretation is parametrized by the thread pool, this is non-trivial even for pure operations! In particular, for reestablishing exploit_wf(\( \rho_\ell, \sigma'_\ell, T'_\ell, \overline{C} \)) for the new thread pool \( T'_\ell \) and state \( \sigma'_\ell \), we need to update the pool_safe(\( \rho_\ell, T'_\ell, [\pi \mapsto e], \sigma'_\ell \)) assertions for exploited locations (\textit{i.e.}, making sure that we can still construct data races for conflicting accesses).

For a pure step in thread \( \pi' \), we consider two cases: if \( \pi \neq \pi' \), we can just replay the step in the alternative configuration of pool_safe(\( \rho_\ell, T_\ell, [\pi \mapsto e], \sigma_\ell \)). (Basically, we are reordering this step around the exploited non-atomic access.) If \( \pi = \pi' \), we can ignore this step since only the \textit{other} threads matter (\( T_\ell, [\pi \mapsto e] \) overwrites the current thread) and the state \( \sigma_\ell \) do not change.

However, the latter case is more complicated for steps that interact with the heap and where possibly \( \sigma'_\ell \neq \sigma_\ell \). Most of these operations can be dealt with via careful extensions of exploit_wf (see Anonymous [2021, §4]). Here we will focus on the most interesting case, which is an atomic store, \textit{i.e.}, maintaining the invariant when proving \textbf{SIM-STORE-SC}. This will also show why this rule is only provable for exploit \( \pi 0 \). First, we can assume that no other thread is exploiting the location \( \ell_\ell \) as there would be a data race otherwise. This means that heapbij contains full ownership of \( \ell_\ell \) and \( \ell_\ell \), which we use to justify the store in source and target. It is also easy to reestablish pool_safe(\( \rho_\ell, T_\ell, [\pi' \mapsto e], \sigma'_\ell \)) for threads \( \pi' \neq \pi \) by replaying the store. However, there is a problem when updating the pool_safe(\( \rho_\ell, T_\ell, [\pi \mapsto e], \sigma_\ell \)) to pool_safe(\( \rho_\ell, T'_\ell, [\pi \mapsto e], \sigma'_\ell \)) for exploited locations of the current thread: the current expression of the thread \( \pi \) in the alternative execution is \( e \) and not the atomic store, so we cannot just mirror the atomic store in the alternative execution. But without this we have a problem as the source state is not in sync any more! The only way out is to require that the current thread does not exploit any locations, which leads to the exploit \( \pi 0 \) precondition of \textbf{SIM-STORE-SC}. For non-atomic stores, we similarly cannot update the state in the pool_safe for locations exploited by the current thread. However, this can be fixed by tweaking the definition of exploit_wf as any thread that would observe this difference in states would race with the non-atomic store.

## 7 SIMULIRIS MEETS STACKED BORROWS

To demonstrate the flexibility of Simuliris as a language-generic framework, we instantiate it with the language of Stacked Borrows [Jung et al. 2020], a memory model for Rust proposed to enable aliasing-based optimizations. In §7.1, we give a brief overview of Stacked Borrows to explain how the correctness proofs of such optimizations benefit from Simuliris. In particular, Simuliris helped a lot when extending the soundness proofs of these optimizations to \textit{concurrency} (with sequentially consistent accesses). We verify correctness not only of the original paper’s optimizations, but also of a new loop hoisting optimization that makes use of Simuliris’s support for coinduction.

\(^{18}\)The full simulation relation uses pool_safe(\( \rho_\ell, T_\ell, \sigma_\ell \)) instead of safe(\( \rho_\ell, e_\ell, \sigma_\ell \)) as shown in §5.
with SimuLang (whole-program adequacy, coinduction, basic structural rules), and that we obtain
an ad-hoc ownership-based coinductive open simulation relation not unlike the model of
unsafe when unsafe structures with heavy aliasing to be implemented in
Simuliris. More specifically, the correctness of an optimization relies on the notion of ownership
tags may access this location; using a reference with a different tag is undefined behavior.
However, this also means that Rust’s type system guarantees are not implicitly upheld any more
when unsafe code is being used. We thus need to explicitly demand that unsafe code follows the
aliasing discipline. Stacked Borrows makes this precise, while still allowing pointer-based data
structures with heavy aliasing to be implemented in unsafe code. The basic idea is to associate
each reference with a tag, and each location with a borrow stack of tags. The stack tracks which
tags may access this location; using a reference with a different tag is undefined behavior.
The original Stacked Borrows formalization verifies several optimizations in a sequential language
with an ad-hoc ownership-based coinductive open simulation relation not unlike the model of Simuliris. More specifically, the correctness of an optimization relies on the notion of ownership
of some tag \( t \). In the example in Figure 12a, when the function gets the argument \( x \) as a mutable
reference to a location \( \ell_x \), it also acquires the ownership \( \text{OwnTag}(t_x) \) of a unique tag \( t_x \) associated
with \( x \). \( \text{OwnTag}(t_x) \) intuitively enforces that the tag \( t_x \) is at the top of \( \ell_x \)’s stack. The correctness
argument then relies on two points: (i) as the function body keeps \( \text{OwnTag}(t_x) \) throughout its
operations, the function \( f \) has no ownership of \( t_x \), and so \( f \) cannot use the tag \( t_x \) to access \( \ell_x \);
and (ii) \( f \) cannot use any other tag to access \( \ell_x \) either, as such an access would pop \( t_x \) from \( \ell_x \)’s
stack, making the load \( \star x \) in the source (right after the call of \( f \)) undefined behavior. Ultimately the
ownership of \( \text{OwnTag}(t_x) \) prevents \( f \) from accessing \( \ell_x \), so moving the load up is sound.

7.1 Stacked Borrows: An Aliasing Model for Rust

Rust is a systems programming language that gives strong static guarantees through an ownership-
based type system and enforces an aliasing principle of Aliasing XOR Mutability (AXM): at any
point in time, data in memory either have one mutable reference \&mut \( \mathcal{T} \) that is unique and allows
mutation, or multiple shared references \&\&\( \mathcal{T} \) that only allow read access.
In principle, these guarantees can enable aliasing-based optimizations, such as the one given
in Figure 12a (“Example 1” in [Jung et al. 2020]). It works on the body of a Rust function whose
argument is a mutable reference \( \star x: \&\text{mut } \mathcal{I} \mathcal{3} \mathcal{2} \) to a 32-bit signed integer, and whose return type is
\( \mathcal{I} \mathcal{3} \mathcal{2} \). As the function takes a mutable reference \( x \), by AXM, the unknown function \( f \) should not have
an alias to the memory location referenced by \( x \). Thus, the call to \( f() \) should not change the value
of \( x \), making it safe to move the load from \( x \) up across the call. (And this in turn should enable
constant propagation of 42 to avoid the load altogether.)
But there is a problem: Rust makes heavy use of unsafe code. The purpose of unsafe code is to let
programmers write code that is correct for reasons that are too subtle for the compiler to understand.
However, this also means that Rust’s type system guarantees are not implicitly upheld any more
when unsafe code is being used. We thus need to explicitly demand that unsafe code follows the
aliasing discipline. Stacked Borrows makes this precise, while still allowing pointer-based data
structures with heavy aliasing to be implemented in unsafe code. The basic idea is to associate
each reference with a tag, and each location with a borrow stack of tags. The stack tracks which
tags may access this location; using a reference with a different tag is undefined behavior.
The use of ownership-based reasoning makes Stacked Borrows an excellent candidate application
of Simuliris. We have instantiated Simuliris with the Stacked Borrows language, derived a logic
for proving optimizations (closely following the original setup), and ported all the optimization
proofs. The place where Simuliris shines is that a lot of the simulation infrastructure can be shared
with SimuLang (whole-program adequacy, coinduction, basic structural rules), and that we obtain

Fig. 12. Two examples of Stacked Borrows optimizations done in Simuliris.
a proper separation logic with an interactive proof mode for carrying out proofs of optimizations without directly reasoning about the underlying model of resources.

Using this infrastructure, it was easy to prove a new loop hoisting optimization (Figure 12b). Here, \( f \) and \( g \) are closures and \( x \) is an immutable shared reference. We have proven that the repeated loads \( *x \) in each iteration can be replaced by a single load before the loop. The intuition is that ownership of the tag \( t_x \) for the shared reference \( x \) can be maintained through the entire loop.

A concurrent version of Stacked Borrows. Jung et al. [2020] only consider a sequential language. We have extended the Stacked Borrows language with support for concurrency and shown that the original optimizations are still correct. As part of this we discovered that a direct port of the original Stacked Borrows semantics, load instructions (e.g., \( *x \)) directly trigger UB when the tag is not in the borrow stack. In a concurrent setting, this choice invalidates moving loads up like in Figure 12a: if the tag is removed from the stack by a concurrent thread, the target would trigger UB before \( f \) is even called, whereas the source would avoid UB entirely if \( f \) never returns!

We have thus relaxed the original semantics: instead of directly triggering UB when doing a load with an invalid tag, the load returns poison, a special value which only triggers UB when it is being used (a form of deferred UB [Lee et al. 2017]). We can elegantly reflect this into separation logic: when the target load from \( \ell \) yields poison, we obtain ownership of the assertion Tainted\((t, \ell)\) stating that a tag \( t \) can never be contained in the source stack for \( \ell \) again. Later in the proof, we use this assertion to prove that the source also loads poison, so both executions are in sync again.

8 RELATED WORK

Simuliris is, to our knowledge, the first separation logic designed for verifying concurrent compiler optimizations. This brings together two broad lines of research, of which we discuss the most closely related work here. Also see Figure 1 in the introduction for an overview.

Concurrent compiler optimizations. As already mentioned in the introduction, there are several projects with the goal of equipping CompCert with support for concurrency. Of course, verifying an actual optimization pass is a much bigger task than just verifying a few particular program transformations in isolation: one needs to show that some algorithm performing the transformation (or some validator checking it) will always produce a suitable simulation between target and source program. In the future, it would be interesting to explore the integration of Simuliris into a compiler correctness proof. Meanwhile, it is not surprising that the actual optimizations performed by CompCert, while impressive, do not reach the same complexity as optimizations that can be verified in isolation, such as in our approach. Indeed, even for purely sequential code, CompCert will not hoist loads from escaped pointers out of a loop. Correspondingly, our main motivating example (§1) is not (to our knowledge) verified by any of these variants of CompCert.

That said, the infrastructure for reasoning about sequential code in CompCert is in some ways more powerful than what we built for SimuLang. For example, CompCert’s “memory injection”, which corresponds to our heap bijection managing escaped locations, supports mapping multiple memory blocks of the source program into a single target block, whereas we enforce a one-to-one mapping of source and target blocks. On the other hand, since our bijection is managed via separation logic ghost state, it is arguably more convenient to work with. We leave it to future work to find a nice separation logic interface for the full power of CompCert’s memory injection.

CASCompCert [Jiang et al. 2019] and Concurrent CompCert [Beringer et al. 2014; Cuellar 2020] show that sequential reasoning can be applied to non-synchronizing fragments of concurrent programs. This should, in principle, be sufficient to verify our main example using sequential reasoning only (except, as already mentioned, CompCert does not perform this optimization...
even for sequential code). However, both of these approaches have a fundamental limitation: the
operational semantics used to reason about program transformations is entirely sequential; all
effects of concurrency are delegated to side-effects of opaque external function calls. This rules
out any optimization for which one has to reason in detail about the behavior of a synchronizing
operation, such as in our example in §3.2.

The Concurrent Abstraction Layers [Gu et al. 2018] variant of CompCertX [Gu et al. 2015] is
(to our knowledge) the only concurrent variant of CompCert that reasons about fairness. Their
“push/pull” model of shared memory is akin to our data-race detector, though we do not need a
“global log”. However, in their work, all interactions with shared memory are handled via external
function calls. This appears to rule out even the most basic optimizations on non-atomic accesses.

CompCertTSO [Ševčík et al. 2013] uses the same approach as ours: they build concurrency
into the operational semantics. However, the TSO (total store order) model of concurrency gives
well-defined behavior to racy programs, and thus does not permit optimizations such as the example
from the introduction.

Besides the work on CompCert, there also has been a lot of work on models of weak-memory
concurrency, and the set of supported program transformations is a relevant point of comparison
in this space. In particular, some of that work verifies correctness of various memory access
reorderings [Ševčík 2011; Morisset et al. 2013; Vafeiadis et al. 2015; Kang et al. 2017]. Those models
often consider a larger subset of C11 atomics (whereas we just support non-atomic and sequentially
consistent accesses). However, all of that work only considers finite program executions, i.e., they
do not verify termination preservation. They also only verify reorderings of immediately adjacent
instructions and do not formally prove that this is sufficient for more complex optimizations such
as hoisting an instruction out of a loop. We have verified all previously established reorderings
that involve only the accesses supported by our system and at least one non-atomic access (i.e., the
reorderings that are related to data races); see our supplementary material for details. We have
also verified one more reordering that has not been verified in prior work: moving a non-atomic
load before a sequentially consistent load. This optimization is correct in our system, but it is
interestingly incorrect in memory models with C11-style relaxed accesses—to account for the
particularly weak behavior of ARM processors (which ignore control dependencies between loads),
the definition of data races around relaxed accesses has the side-effect that this optimization can
introduce a race into a previously race-free program, rendering it invalid.

Separation logic for contextual refinement. The idea of using separation logic reasoning in
a simulation relation has been explored before, but mostly in the context of using refinement for
program verification: the goal is to show that an implementation of an abstract data type implements
a specification. As such, this line of work lacks the ability to exploit undefined behavior, as we
require for our example. However, that is not the only limitation.

The line of work by Liang et al. [2014]; Liang and Feng [2016] on rely-guarantee style relational
separation logics supports concurrency and fair termination preservation. However, it does not
support passing pointers between the context and library (as would be the case in our motivating
example). Unlike Simuliris, this work supports blocking operations—however, to ensure fair termi-
nation preservation, the user has to reason explicitly about decreasing metrics, which is entirely
hidden in Simuliris via implicit stuttering. Moreover, this work has not been mechanized.

Another family of work uses Iris [Jung et al. 2015, 2016, 2018b; Krebbers et al. 2017a], a framework
for defining concurrent separation logics with a flexible form of “ghost ownership”, as the logical
basis for refinement proofs. In particular, ReLoC [Frumin et al. 2018] shows how to define a binary
logical relation in the (usually unary) Iris program logic, and establishes a number of contextual
refinement results. However, Iris uses the technique of step-indexing to define higher-order ghost
state, which much prior work uses to great effect, but which comes with a serious downside: step-indexing implies that Iris can only be used to reason about safety properties, not liveness properties such as termination preservation. In ReLoC, a diverging implementation refines all specification programs. Tassarotti et al. [2017] bend that limitation by showing that Iris can in fact be used to establish fair termination preservation under the assumption that the non-determinism in the language under consideration is finitely bounded. Recent work by Spies et al. [2021] shows that transfinite step-indexing permits the use of Iris for verifying liveness properties such as termination without such an assumption, but this work has so far only been applied to sequential programs. In our work, we avoid the need for step-indexing in the first place, thus overcoming the limitation of bounded non-determinism while still supporting concurrency without the complexities of transfinite step-indexing. In the future, it would be interesting to explore which new reasoning principles can be obtained in Simuliris by making use of transfinite step-indexing.

REFERENCES


