Pontifícia Universidade Católica
do Rio de Janeiro

Renan Almeida de Miranda Santos

Revisiting Monitors, Again

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática, do Departamento de Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
April 2025

Pontifícia Universidade Católica
do Rio de Janeiro

Renan Almeida de Miranda Santos

Revisiting Monitors, Again

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática. Approved by the Examination Committee:

Prof. Roberto Ierusalimschy
Advisor
Departamento de Informática – PUC-Rio

Profª. Anamaria Martins Moreira
Universidade Federal do Rio de Janeiro – UFRJ

Prof. Hugo Musso Gualandi
Universidade Federal do Rio de Janeiro – UFRJ

Prof. Francisco Figueiredo Goytacaz Sant'Anna
Universidade Estadual do Rio de Janeiro – UERJ

Drª. Noemi de La Rocque Rodriguez
Pesquisadora Autônoma

Rio de Janeiro, April 25th, 2025

Renan Almeida de Miranda Santos

Earned his Bachelor's and Master's degrees in Computer Science from Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

Bibliographic data

Santos, Renan Almeida de Miranda

Revisiting Monitors, Again / Renan Almeida de Miranda Santos; advisor: Roberto Ierusalimschy. – 2025.

68 f: il. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2025.

Inclui bibliografia

1. Informática – Teses. 2. Linguagens Para Programação Concorrente – Teses. 3. Monitores. 4. Condições de Corrida. 5. Semântica Operacional. 6. Provas Formais. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I am deeply grateful to my advisor for the countless meetings, insightful discussions, and thorough reviews.

I would also like to thank my family for being my safe harbor.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

This study was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brasil (CNPq).

Abstract

Santos, Renan Almeida de Miranda; Ierusalimschy, Roberto (Advisor). **Revisiting Monitors, Again**. Rio de Janeiro, 2025. 68p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Over the years, many programming languages were proposed with the goal of being free from data races. The concept of monitors, in particular, was envisioned precisely to prevent data races. In this thesis, we describe a data-race-free programming language inspired by monitors. Our languages features threads, shared memory with referential semantics, unbounded loops, and a novel construct that emulates monitor capabilities. This new construct is interesting because it is simpler and at the same time more flexible than monitors. We prevent data races in the language by using a type system that allows only immutable or protected data to be shared among threads. We defined the small-step operational semantics of this language through three distinct and hierarchically layered single-step operations: roughly, the first layer represents individual cores, the second layer manages the shared memory, and the third layer manages the thread pool. Over these relations, we defined a multistep relation that produces a trace of program execution. We use this trace to prove the absence of data races. All these results were formally defined and proved in Coq.

Keywords

Monitors; Data Races; Operational Semantics; Formal Proofs.

Resumo

Santos, Renan Almeida de Miranda; Ierusalimschy, Roberto. **Revisitando Monitores, Novamente**. Rio de Janeiro, 2025. 68p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Ao longo dos anos, muitas linguagens de programação foram propostas com o objetivo de prevenir condições de corrida para acessos à memória. O conceito de monitores, em particular, foi concebido justamente para evitar condições de corrida com acessos à memória. Nesta tese, nós descrevemos uma linguagem de programação inspirada por monitores e livre desse tipo de condição de corrida. Nossa linguagem possui threads, memória compartilhada com ponteiros, laços não limitados, e uma construção nova que emula as capacidades de monitores. Essa nova construção é interessante porque é mais simples e, ao mesmo tempo, mais flexível do que monitores. Nós prevenimos condições de corrida na linguagem por meio de um sistema de tipos que permite apenas o compartilhamento entre threads de dados imutáveis ou protegidos. Nós definimos a semântica operacional em passo pequeno da nossa linguagem por meio de três operações distintas e hierarquicamente organizadas: grosseiramente, a primeira camada representa núcleos individuais, a segunda camada gerencia a memória compartilhada e a terceira camada gerencia o conjunto de threads. Sobre essas relações, nós definimos uma relação de múltiplos passos que produz um traço da execução do programa. Nós usamos esse traço para provar a ausência de condições de corrida. Todos esses resultados foram formalmente definidos e provados em Coq.

Palavras-chave

Monitores; Condições de Corrida; Semântica Operacional; Provas Formais.

Table of contents

1
Introduction

It is fair to say that most mainstream devices of this era have CPUs with multiple cores. Parallelism as a means to increase performance has become ubiquitous, and we cannot ignore its impact on the way we develop programs. In specific, programming languages that combine preemptive multithreading with the shared memory model impose on programmers a whole new class of time-dependent errors that are specially hard to deal with [14, 15]. This thesis is about data-race errors and programming languages designed to avoid them.

The concept of monitors have been created precisely to prevent data races. Monitors encapsulate shared variables and restrict their access through monitor procedures, which themselves are always called in mutual exclusion [9, 10]. Concurrent Pascal [11], the first language to implement monitors, takes advantage of the guarantees provided by the model to statically check for the absence of race-prone code. In our previous work [2, 3], we revisited the concept of monitors and applied it to a programming language with pointers, a powerful feature that was not present in the original proposals. In the present work, we once again look into monitors as a way to implement data-race-free languages, but now focusing on the formal aspects of this proposal. An important result of this process was the decomposition of monitors in its essential components.

The work of formally proving that a language satisfies some property begins by defining a formal semantics for the language: if we are to prove that something is true when a program runs, we need to know what "runs" entails. Formalizing a language property is also not a trivial matter, specially for data race freedom. A data race is an event that happens during the execution of a program, thus, we define its absence in terms of all the possible sequences of steps a program can take. The very notion of this definition is dependent on the formal semantics of a language.

In order to be able to prove that a language is data-race free, we need the language to enforce this characteristic. There are many ways to do that, ranging from Rust's ownership type system to pure functional languages that restrict themselves to immutability. As discussed, our inspiration is the monitor concept, so we chose to prove data race freedom for a language that guarantees that mutable data being shared between threads always does so in mutual

exclusion.

To illustrate the safety guarantees behind the essence of the monitor model, we created Elo, a toy concurrent language with threads that share data. Elo provides mutual exclusion capabilities through its single non-conventional feature: *monitored references*. Unlike standard references, monitored references cannot be freely read from or written to. Accessing a monitored reference requires entering its critical region, which is done in mutual exclusion with any other threads trying to do the same. Thus, monitored references provide the same safety guarantees that monitors do, without, however, its class-like modularity features. With that, we decompose monitors into its most basic components. Besides monitored references, Elo has enough features so we can measure it against full-fledged languages: it has basic arithmetic operations, conditionals, loops, functions as first-class values, memory manipulation operations (read, write, and allocation), `wait` statements, and `spawn` statements for creating threads.

In addition to monitored references, Elo also has read-only and writeable references. As the name implies, read-only references can only be read from, while writeable references can be read from and written to. Elo's type system enforces safety by restricting how writeable references (which are the prime material of data races) can be shared between threads. Roughly, Elo ensures that writeable references shared by two or more threads are always accessed in mutual exclusion. Despite the apparent simplicity for a race-free guarantee in this scenario, trying to formally prove it was challenging, and we did not find techniques and examples in the literature that we could use as a basis. One of the main contributions of this thesis lies in showing the rationale of that proof.

As mentioned, we need to provide an operational semantics for Elo in order to be able to prove data race freedom. Normally, we would do so by defining a single reduction relation that describes how a memory and a pool of threads get transformed by a single step of execution. We could formalize this relation by an inductive set of rules. This is the basic idea behind small-step operational semantics. Note, however, that several rules for these reduction steps would not care about the memory, and most would not care about the thread pool per se, only about a single thread. For example, the semantics behind function calls is oblivious both to the memory and to the thread pool. In order to improve modularity, Elo's small-step rules are decomposed into three hierarchical step operations: term step, memory step, and concurrent step. Term step concerns itself solely with how a term (a thread) reduces, memory step adds the shared memory semantics, and concurrent step adds multithreading. By doing this, we avoid repeating unnecessary information in

our definitions.

We can think of our three step operations as components in the multi-core architecture of computers: term step represents a core, memory step represents the memory controller unit, and concurrent step represents the thread pool controller inside the operating system. The steps interact with each other through *effects*, which act as the bus in our analogy. They describe the *side effects* of the operation being performed by the reductions and serve as a channel for relevant data to be exchanged between step levels. For example, the READ effect describes the act of reading from the memory. It allows the address from the term and the loaded value from the memory to be passed between the CPU level and the memory level.

Effects also act as *labels* for our step operations, as in a *Labelled Transition System* [20]. In that regard, the language's multistep operation uses effects to produce a trace of the execution of the program. A trace is a sequential list of events, with each event being composed of a thread identifier and and effect. In essence, this list describes which thread caused which effect for each point in the discrete timeline of the program's execution. Effects, traces, and the hierarchy of steps allowed us to prove language properties in a legible and modular manner.

Our main safety result states that, for all possible traces that an Elo program can produce, there is no possible scenario in which two threads perform memory operations in a manner that would create a data race. For the concept of data races, we adopt the same definition of the Java [16] and C++ [17] memory models. We say that two memory operations on the same memory location form a data race if at least one of them is a write operation, and they do not follow a *happens-before* order. Two operations are ordered by a happens-before relation if both execute on the same thread or if they synchronize over the same lock.

To achieve this result, we define some auxiliary properties and prove that they hold for all the steps of a program's execution. We call these properties invariants. There are two core invariants to the overhaul schema of our safety proof: one deals with mutual exclusion and the other tracks shared data between threads. Elo uses standard locks to implement mutual exclusion, so the first invariant states that when a lock is locked, then one and only one thread is inside its critical region, and when a lock is unlocked, then no threads are inside its critical region. The second invariant uses *regions* in order to group memory locations. A thread can be in its own unique region or inside a monitored reference region. The invariant ensures that if a reference is inside a region, than it is *owned* by that region. This allows us to assign *exclusive ownership*

to memory locations, meaning an address is either owned by a specific thread or by a monitored reference. Ownership of this kind does not change during execution; if an address is allocated and begins its life cycle belonging to a thread, it will forever be attached to that thread. Moreover, if an address can access another address, then both addresses belong to the same owner. As we will see, we are able to prove safety by combining this notion of regions and ownership with the mutual exclusion guarantees of the first invariant.

We formalized our definitions and checked our proofs in Coq [18] (8.20.0), a proof management system. You can check the Coq code at [1].

The rest of this thesis is organized as follows. In Chapter 2, we discuss the language and its features. In Chapter 3, we formalize Elo's syntax, operational semantics, and typing rules. In Chapter 4, we discuss the main properties behind our proofs. In Chapter 5, we present the main safety theorem and explain how we proved it. In Chapter 6, we discuss related work. Lastly, in Chapter 7 we present our concluding remarks.

2
The Language

In this chapter, we introduce Elo, the programming language we created to use in our formalizations. Elo is an expression-oriented[1] concurrent programming language with mutable memory and threads that can share data. We designed Elo to formalize memory operations in a safe threaded environment; thus, the language only includes features relevant for providing sufficient expressivity and proving interesting properties regarding memory safety and concurrency.

Elo is meant to be simple. We tried to break the semantics of the language into basic constructs and to assign a single concept to each of them. This approach to simplicity is quite common in programming-language research. For instance, in most languages variables are mutable by nature (hence their names), however, representing values with identifiers and having mutable values are two different concepts. For this reason, as we will see, we separated them as several functional languages do. We do something similar with monitored references. Their responsibility is to provide synchronized access to a value, not to provide mutability capabilities; thus, monitored references cannot be written to. It is up to the programmer to put a mutable reference inside a monitored reference in order to implement mutable synchronized state. This portents to the fact that Elo is closer to a high-level intermediary representation than to a general purpose programming language. Elo's goal is to provide a set of features concise enough to keep our proofs simple, but wide enough so we do not weaken their merits. Some of the usual features that are missing from the language, such as arrays and records, could be implemented through grunt work. Also, we express some standard high-level constructs in Elo by combining one or more of its own low-level constructs.

Before we delve into the specifics of the language, let us first familiarize ourselves with its concepts by analyzing two examples of programs written in Elo. The first example implements a calculator for the Fibonacci sequence and showcases most of Elo's standard constructs. The second example explores the concurrency features of the language by tackling the producer-consumer problem.

[1] Every one of Elo's constructions is an expression and thus reduces to a value.

```
1  function fib(n : Nat) : Nat
2    if n < 2 then
3      n
4    else
5      let a : w&Nat = new w&Nat 0 in        // first element
6      let b : w&Nat = new w&Nat 1 in        // second element
7      let x : w&Nat = new w&Nat 1 in        // result
8      let i : w&Nat = new w&Nat (n - 2) in // counter
9      while !i do
10       a := !b;       // updates the first element
11       b := !x;       // updates the second element
12       x := !a + !b; // updates the result
13       i := !i - 1    // decreases the counter
14     end;
15     !x // returns the result
16   end
17 end
18 fib(10) // yields 55
```

Figure 2.1: Calculating the Fibonacci sequence.

Figure 2.1 presents the implementation for the Fibonacci calculator. The `fib` function receives the index of an element in the Fibonacci sequence and returns the value of said element. The function first checks for the base cases with an `if` statement, then iteratively calculates the Fibonacci sequence by using a `while` statement.

Elo follows ML's style in which variables are always immutable and mutability only occurs in memory cells. The variable `a` from line 5 evidences this feature, as it is a `let` constant that points to a mutable memory cell. (The `b`, `x`, and `i` variables follow the same reasoning.) The expression ⟨`new w&Nat 0`⟩ allocates a memory cell and initializes it with zero; then, the variable `a` gets assigned the reference that points to that cell. The type ⟨`w&Nat`⟩ in the `new` expression means "writeable reference to a number"; it indicates which type of reference is being created. Besides writeable references (`w&`), Elo has read-only (`r&`) and monitored (`x&`) references. We will discuss the distinctions between these three types later.

As shown in lines 9 through 13, we retrieve the value stored in a memory cell by dereferencing a reference with the ⟨`!`⟩ operator. In a similar manner, we store a new value in a memory cell with the assignment statement. Assignments expect a reference in the left-hand side of the ⟨`:=`⟩ operator, and a value with a matching type in the right-hand side. With `let`, `new`, dereferencing, and assignment, we emulate conventional variables from imperative languages as (constant) variables that point to (mutable) cells.

Note that you cannot allocate a reference without giving it an initial value, meaning that there are no empty references or null pointers in the

```
1  let buffer : x&(w&Nat) = new x&(w&Nat) (new w&Nat 0) in
2  let max : Nat = 10 in
3  spawn // producer
4    while 1 do
5      acquire buffer (λ n .
6        await (!n < max); // wait until not full
7        n := !n + 1
8      )
9    end
10 end;
11 spawn // consumer
12   while 1 do
13     acquire buffer (λ n .
14       await (!n > 0); // wait until not empty
15       n := !n - 1
16     )
17   end
18 end
```

Figure 2.2: A simplified producer-consumer scenario.

language. Also, there are no booleans in Elo; as in C, conditions are false when they evaluate to zero, and are true when they evaluate to any other number. For this reason, we only need to dereference the counter in line 9 to check if it reached zero.

Figure 2.2 presents a simplified producer-consumer program. It spawns a producer thread and a consumer thread that manipulate a shared buffer. In this contrived version, the buffer holds a number that gets incremented (produced) and decremented (consumed).

The example showcases Elo's main concurrency feature: monitored references. Monitored references closely relate to monitors, as they can be used to implement the concurrent aspects of monitors. We will discuss their relationship in Section 2.3. Going back to the example, note the type of the `buffer` variable: a monitored reference (`x&`) that points to a writeable reference (`w&`) that points to a number (`Nat`). We define `buffer` as a monitored reference in order to synchronize thread access to its contents, and to protect its inner mutable state. As with read-only and writeable references, a monitored reference points to a memory cell, but, unlike other reference types, the language ensures that the contents of monitored cells can only be accessed by a thread in mutual exclusion with other threads. (Let us momentarily skip the discussion on how Elo enforces this safety guarantees, and instead focus on the constructions surrounding monitored references.) The inner mutable reference also hints to the fact that monitored references by themselves are not mutable. As mentioned, just like with variables, we embed mutability into monitored references by having them hold writeable references. Thus, while we cannot

```
1  type : 'Unit'          // unit type
2       | 'Nat'           // number type
3       | 'r&' type       // read-only reference type
4       | 'x&' type       // monitored reference type
5       | 'w&' type       // writable reference type
6       | type '->' type  // function type
7
8  exp : 'unit'                          // unit literal
9      | n                               // numeric literal
10     | exp '+' exp                     // plus
11     | exp '-' exp                     // monus
12     | exp ';' exp                     // sequencing
13     | IF exp THEN exp ELSE exp END    // conditional
14     | WHILE exp DO exp END            // loop
15     | x                               // variable
16     | 'λ' a ':' type '.' exp          // function abstraction
17     | exp '(' exp ')'                 // function call
18     | SPAWN exp END                        // thread creation
19     | NEW type exp                         // allocation
20     | '!' exp                              // dereferencing
21     | exp ':=' exp                         // assignment
22     | ACQUIRE exp '(' 'λ' a '.' exp ')'    // acquire
23     | AWAIT exp                            // await
```

Figure 2.3: Concrete syntax (types and expressions).

assign a value to a monitored reference, we can read its contents and assign a value to its inner writeable reference.

As seen in lines 5 and 13, we must use the `acquire` expression instead of the previously mentioned ⟨ `!` ⟩ operator to read the contents of a monitored reference. An `acquire` expression acquires the lock associated with the monitored reference, reads the contents of the monitored reference from the memory, then calls the attached anonymous function with the read value as the argument. In our example, the producer and the consumer try to acquire the `buffer` to gain access to the inner writeable reference (which gets stored in the variable `n`) and increment/decrement it. An `acquire` also implicitly releases its lock at the end of the called function, essentially delimiting a syntactical critical region for handling a monitored reference.

Critical regions can contain `await` operations that block until a certain condition is met (without holding to the lock while waiting). We see this in line 6, as the producer blocks while the buffer is full, and in line 14, as the consumer blocks while the buffer is empty.

2.1 Syntax

Having reviewed some general examples of Elo programs, let us now focus on its constructs. Figure 2.3 presents the complete concrete syntax for

Elo. Uppercase items and items delimited by single quotes denote terminals (tokens), while lowercase items (`type` and `exp`) represent non-terminals.

Elo's primitive types comprise `Unit` (the type of the `unit` value) and `Nat` (the type of numeric values), with `Nat` serving as an example for other trivial types that are not in the language such as integers, floats, and booleans. Besides the primitive types, the language has function types and three kinds of reference types. The three reference types differentiate from one another by which operations can be performed on them: a read-only reference can only be read from, a monitored reference can only be acquired, and a writeable reference can be read from and written to. Conceptually, monitored references are repurposed regular references that include synchronization features. Instead of creating a new construct, we chose to implement synchronization primitives using references precisely so we can be economic with the constructions we introduce to the language. By doing it this way we reuse most of the semantic constructions meant for memory expressions, which simplifies the proofs and makes Elo more concise.

In order to guarantee data-race-freedom, Elo distinguishes between safe and unsafe types, the idea being that values with safe types can be freely shared between threads. `Unit` and `Nat` are naturally safe. Read-only references are also safe, but only because they must be recursively safe, meaning a read-only reference can only point to values with safe types. This prohibits the existence of a read-only reference that contains a writable reference, which would allow for potential races due to pointer indirection. Monitored references are also safe, since its contents can only be manipulated while in mutual exclusion with other threads. Writeable references are naturally unsafe, and functions are unsafe due to their dependencies on variables from enclosing scopes (closures). We will explain these ideas more thoroughly when we discuss the safety rationale and the type system.

Regarding expressions, a `unit` represents a useless value, mainly the result of operations that only perform side effects, and an `n` represents numeric literals. Arithmetic operations, sequencing, conditionals, and loops work as expected; as we mentioned, conditions inside conditionals and loops compare a number to zero. We could have trivially added multiplication, division, comparisons, and other common operators to the language. We chose not to do so because these constructions would only repeat what addition and subtraction (monus[2]) already exemplify in the proofs. This is in line with what we explained about the goals of the language.

[2]The *monus* operation subtracts two numbers without yielding negative values. If the second value in the expression is greater than the first, then the expression evaluates to zero.

As to functional constructs, an `x` represents a variable, and function abstractions $\langle \lambda\, a : \tau\,.\, t \rangle$ introduce variables; the a stands for the parameter name, the τ stands for the type of that parameter, and t is the body of the function. Functions only take a single parameter, but we can use currying to model multiple parameters. Function application is standard, and basically performs variable substitution, as we will see when we delve into the operational semantics. The `let` construct from the examples is not present in the syntax because we implement it indirectly as a syntactic sugar using function abstraction and function application:

$$\texttt{let}\ a\ :\ \tau\ \texttt{=}\ t'\ \texttt{in}\ t \quad \equiv \quad (\lambda\, a : \tau\,.\, t)\ t'$$

The `function` construct we used to define `fib` in the first example is also a syntactic sugar. Since Elo has functions as first class values, top-level function definitions are simply `let` definitions of anonymous functions:

$$\texttt{function}\ f\texttt{(}\,a\ :\ \tau_a\,\texttt{)}\ :\ \tau\ \ t\ \texttt{end}\ t' \quad \equiv \quad \texttt{let}\ f\ :\ \tau_a \to \tau\ \texttt{=}\ (\lambda\, a : \tau_a\,.\, t)\ \texttt{in}\ t'$$

A `spawn` creates a new thread that evaluates the expression within the body of the `spawn`.

Regarding expressions that concern the memory, the `new` operator allocates and initializes references. It takes both an expression and a type: the expression is used to initialize the corresponding memory cell; the type indicates which kind of reference is being created: a read-only, monitored, or writeable reference. Dereferencing (also called a *load* operation) takes a reference and returns its value from the memory, while assignment takes a reference and a value and writes that value into the reference's memory cell.

An `acquire` operation is technically a memory operation, since it manipulates monitored references, but it mainly works as a synchronization construct. It takes a monitored reference and a function, and goes through four steps. It blocks until it acquires the monitored reference's lock, then it reads its contents, calls the function with the read value as the argument, and, finally, when the function finishes its execution, the `acquire` releases the monitored reference. As mentioned, an `acquire` delimits a syntactical critical region for handling a monitored reference. It also implicitly introduces a `self` variable to the scope of this critical region. The `self` variable is an alias to the acquired monitored reference itself.

An `await` operation can only be performed inside a critical region. It blocks its thread until its condition is met, without retaining the lock of the monitored reference. For comparison, standard monitors use two strategies to define the semantics of its wait operations [13]: *explicit* signal monitors use

```
1       while not (some condition) do
2           release the lock;
3           wait until another thread releases the lock;
4           acquire the lock
5       end
```

Figure 2.4: Pseudocode implementation for the `await` expression.

wait and *signal* constructs paired with condition variables, so that when one thread blocks in a *wait*, another thread has to explicitly signal to restart it; *implicit* signal monitors synchronize using *wait-until* constructs, that block the current thread while a given logical predicate is not true. There are tradeoffs regarding performance and ease of use between the two strategies, with implicit signal monitors usually being slower but simpler, since they eliminate the need for cooperation logic between threads. Nevertheless, the two models are equal in terms of expressivity, so we chose to implement implicit-signaling for Elo because we found it easier to formalize. Concretely, if the condition in an `await` expression is false, it releases the lock and blocks until another thread releases the lock. Then, the `await` tries to reacquire the lock and, once it does, it checks the condition again. If the condition is true, `await` does nothing and reduces to `unit`. Figure 2.4 illustrates this semantics in pseudocode.

2.2 Safety guarantees

In order to guarantee the absence of data races, Elo enforces that writeable references only have one *owner* during their entire lifetimes. Conceptually, a writeable reference in Elo can either belong to a thread or to a monitored reference. A writeable reference that belongs to a thread can only be accessed by that thread, while a reference that belongs to a monitored reference can only be accessed inside the body of a corresponding `acquire` expression. Our data-race-freedom guarantee comes from the fact that thread owners by definition only access their references sequentially, and writeable references that belong to monitored references can only be accessed in mutual exclusion imposed by `acquire` expressions.

We use the syntactic concept of *boundaries* to enforce exclusive ownership for writeable references. A `spawn` statement and both the initialization and the `acquire` of a monitored reference each delimit a boundary. The type system prevents values with unsafe types from crossing these boundaries by restricting variable visibility from enclosing scopes and by making sure that values returned by `acquire` expressions do not have unsafe types. Thus, writeable references (which are unsafe) do not gain or switch owners.

```
1 let a : w&Nat = new w&Nat 42 in
2 spawn
3     a := 0 // error: undefined variable ‘a’
4 end;
5 a := 1
```

Figure 2.5: Variable visibility for `spawn` statements.

```
1 let a : w&Nat = new w&Nat 1337 in
2 let m : x&w&Nat = new x&w&Nat (
3     a // error: undefined variable ‘a’
4 ) in
5 spawn
6     acquire m (λ n . n := 0)
7 end;
8 a := 1
```

Figure 2.6: Variable visibility for (monitored) initializers.

```
1 let m : x&w&w&Nat = new x&w&w&Nat (
2     new w&w&Nat (new w&Nat 2)
3 ) in
4 let a : w&Nat = new w&Nat 1 in
5 acquire m (λ n . n := a); // error: undefined variable ‘a’
6 spawn
7     acquire m (λ n . !n := 0)
8 end;
9 a := 1
```

Figure 2.7: Variable visibility for `acquire` expressions.

Let us illustrate the aforementioned restrictions for boundaries using some examples. Figure 2.5 shows why we cannot let an unsafe value cross `spawn` boundaries. In line 1 we define a writeable reference `a` and try to share it with the thread spawned in line 2. The language yields an error since `a` is not visible inside the `spawn` statement. If we did not have that check, then the assignments in lines 3 and 5 would form a data race.

Figure 2.6 showcases why we cannot use unsafe values when initializing monitored references. In line 1 we create a writeable reference and store it in the variable `a`, and in lines 2 through 4 we try to wrap it with a monitored reference. As the variable is not visible within the scope of the `new` expression, we get an error. Without it, the assignments in lines 6 and 8 would concurrently access the same writeable reference, and thus cause a data race. Intuitively, unsafe values manipulated within the initializer of a monitored reference are *owned* by that monitored reference, as they can later appear inside an `acquire` expression. For this reason, the variable `a` cannot belong to both the thread and the initializer of the monitored reference.

```
1 let m : x&w&Nat = new x&w&Nat (new w&Nat 24) in
2 let a : w&Nat = acquire m (λ n .
3    n  // error: unsafe type
4 ) in
5 spawn
6     acquire m (λ n . n := 0)
7 end;
8 a := 1
```

Figure 2.8: The value of an `acquire` expression

The visibility rule for `acquire` expressions follows the same logic. Figure 2.7 illustrates this concept. In line 1 we create a monitored reference `m` that wraps a writeable reference that points to another writeable reference. In line 4 we define a new writeable reference `a`, and in line 5 we try to assign it to the inner writeable reference of `m`. Since `a` cannot cross the `acquire` boundary, we get an error. If we did not have that restriction, we would have the same problem of Figure 2.6, so the assignments in lines 7 and 9 would form a data race.

Figure 2.8 highlights the rule that enforces that the value returned by an `acquire` expression must be safe. In line 1 we create a monitored reference with an inner writeable reference. In line 3 we try to escape that writeable reference from the monitored reference using an identity function. This yields an error because `n` has type ⟨ `w&Nat` ⟩, which is unsafe. If not for this error, the assignments in lines 6 and 8 could cause a data race.

In short, these examples illustrate the safety guarantees embedded in Elo's typing system. As we will see, the restrictions in the typing rules reflect these cases. For example, the typing rule for the expression ⟨ `spawn` *exp* ⟩ types the body of the spawn using an environment that excludes mutable variables, so that any attempt to use an external mutable variable raises a type error. Also note that the fact that we were able to prove the absence of data races for well-typed Elo programs is the definitive argument for the exhaustiveness of the rules. If there was a missing restriction, Coq would not allow us to complete the proof.

2.3 Comparison with monitors

Monitored references can do everything that monitors can and more while still maintaining the same guarantees of the model. Before we compare the two, let us recapitulate the essence of the monitor concept.

Monitors were proposed by Brinch Hansen [9] and Hoare [10]. A monitor is a class-like data structure that contains encapsulated variables and a

selection of methods that can only be called in mutual exclusion between multiple threads. In more abstract terms, a monitor provides a pre-defined interface for programs to safely interact with data shared between threads. Monitor methods may also block while waiting for a specific change of state inside the monitor.

The original monitor proposals were not envisioned for languages with references [29]. The rule that prevented race conditions was a simple scope rule that only permitted a method to access their own variables or the monitor's variables. In a language with pointers, it is obvious that such guard is not enough to guarantee data race freedom. For example, imagine a monitor that stores a reference to an integer with a method that returns this pointer. Similar to the examples we discussed in Section 2.2, such configuration would allow the method to "leak" protected data from the monitor. We addressed this issue in our first time revisiting the monitor concept [3]. We implemented Aria, a language with references and monitors that maintained the model's safety guarantees.

Aria is a standard imperative programming language with native threads that share data with one another. Aria uses monitors and the concept of immutability to guarantee the absence of data races, since mutable shared data can only be accessed inside monitor operations. The implementation of references in Elo draws much from these ideas. Particularly, the restrictions that Aria's type system imposes on the boundaries between threads and monitors are very similar to those that Elo enforces for its threads and monitored references. With Aria, we also expanded on the concept of monitors with scoped permissions and unlocked monitors. Scoped permissions allowed monitors to express resource synchronization using syntactical scopes, while unlocked monitors aimed at dealing with the issue of constantly acquiring and releasing locks when repeatedly calling monitor methods.

To illustrate the problem unlocked monitors were trying to solve, consider a monitor with basic `get` and `set` methods that encapsulates an array of strings. Now imagine the common task of iterating over such array. We have to lock and unlock the monitor for each `get` and `set` call; this repetition can get tremendously expensive. Our solution in Aria involved creating a construct for unlocking a monitor within a syntactically defined region, similar to what Elo's `acquire` does. However, because we were still restricted by the monitor syntax and representation, the implementation for that feature relied on dynamic checks, which is undesired. The deconstruction of monitors into monitored references is the evolution of this idea, as we can check the rules for monitored references statically.

Another important concept associated to monitors is reentrancy. For example, when a thread calls a monitor method that calls another method from the same monitor, it is unclear whether the thread blocks (in a deadlock) or not. Such scenarios are not uncommon, and it is somewhat undesired that the semantics in these cases is not explicitly set by the original definitions of the concept. For languages that use locks to implement monitors, in particular, what defines the semantics is the reentrancy capabilities of said locks. As we will see, the semantics of reentrancy is clearer for monitored references.

It is also interesting to analyze how the idea behind monitors is usually misrepresented in other programming languages. For example, in Java, the *Object* superclass (from which all other classes descend) comes with a lock and a condition variable. Those features, combined with the *synchronized* keyword and the *wait* and *notify* methods, are sometimes enough for some authors [28] to characterize it as a monitor implementation. However, this reflects a lack of understanding of the original monitor proposals. A monitor is not a collection of features from which the programmer can code their own data race freedom. The whole point of the construct is the safety ensured by the language, which is precisely the reason for why we are once again studying it.

Regarding the comparison between monitors and monitored references, note that a monitored reference, just like a monitor, also encapsulates data and provides waiting capabilities through `await`. Monitored references, however, do not statically set how the rest of the program can interact with that data. Thinking abstractly, monitors pair safety guarantees with class-like modularity features, while monitored references forgo the latter to focus on the former. This is not to say that monitored references are limited if compared to monitors; on the contrary, the `acquire` expression allows monitored references to be handled dynamically, without the need for pre-defined functions. Thus, the disassociation of safety from orthogonal modularity concerns expands on the expressivity of monitors.

As to the missing class-like capabilities, Elo does not have object-oriented features mainly because implementing them is orthogonal to the concurrency properties we want to prove. Adding such constructs to the language would fall into the grunt work category we mentioned earlier. That being said, we could implement monitors using monitored references in Elo if we added something like records to the language. Because Elo has first-class functions, we can set the fields of a record to represent both monitor variables and methods. Thus, we can use such records wrapped by monitored references to mimic monitors.

Note that this implementation with records eliminates some of the issues with method reentrancy we mentioned earlier. For example, when recursively

calling a monitor method, we do not need to worry about locks because we can use the record instead of the monitored reference to call the method. The semantics of this type of usage is very clear. It is obvious that calling any record method will not block, since anonymous functions stored within records have nothing to do with locks or concurrency.

3
Formalization

In this chapter, we formalize Elo's syntax, operational semantics, and typing system.

3.1
Syntax

Figure 3.1 lists the terms of Elo's abstract syntax. We implemented them in Coq using the inductive type `tm` in the `Sem.v` file along with the other definitions of the semantics[1]. We use "terms" here instead of "expressions" to differentiate the formal syntax from the concrete syntax. Most of the terms in the list are identical to their expression counterparts. The extra terms (marked with an asterisk) were added to the language so as to express the intermediary results of evaluating some operations.

In order to better understand why we added the new terms, let us quickly review the logic behind small-step operational semantics [20]. The main idea behind small-step semantics is that we break the evaluation of a program, as the name implies, in small steps, with each step performing a syntactical transformation (a reduction) to the term that represents the program. To illustrate, let us analyze the following reduction scheme.

$$\begin{aligned} (4+3)+(2+1) &\xrightarrow{\text{step}} \\ 7+(2+1) &\xrightarrow{\text{step}} \\ 7+3 &\xrightarrow{\text{step}} \\ 10 & \end{aligned}$$

The evaluation of $\langle\,(4+3)\ +\ (2+1)\,\rangle$ is composed by three steps: the first step reduces the subterm $\langle\,4\ +\ 3\,\rangle$, which syntactically transforms the initial term into $\langle\,7\ +\ (2+1)\,\rangle$; the second step reduces the subterm $\langle\,2\ +\ 1\,\rangle$, which yields $\langle\,7+3\,\rangle$; the third step performs the final addition, which ends the evaluation since 10 is a value (a term that cannot be further reduced). Different

[1]The text "inductive type `tm` in `Sem.v`" in the caption for Figure 3.1 indicates this information. Other figures will display information about the Coq definitions in a similar manner.

t	:=		TERMS
		`unit`	*unit literal*
	|	n	*numeric literal*
	|	t_1 `+` t_2	*arithmetic plus*
	|	t_1 `-` t_2	*arithmetic monus*
	|	t_1 `;` t_2	*sequencing*
	|	`if` t_1 t_2 t_3	*conditional*
	|	`while` t_1 t_2	*loop*
	|	x	*variable*
	|	$\lambda\, a : \tau\, .\, t$	*function abstraction*
	|	t_1 t_2	*function application*
	|	`new` τ t	*memory allocation*
	|	`init` τ ad t	* *memory initialization*
	|	$ad : \tau$	* *reference literal*
	|	`!` t	*memory read*
	|	t_1 `:=` t_2	*memory write*
	|	`acq` t_1 `(`$\lambda\, a\, .\, t_2$`)`	*acquire operation*
	|	`cr` ad t	* *critical region*
	|	`wait` t	*wait operation*
	|	`reacq` ad	* *reacquire operation*
	|	`spawn` t	*thread creation*

Figure 3.1: Abstract syntax – terms (inductive type `tm` in `Sem.v`).

from a conventional language interpreter, that reads a program as a set of instructions to be performed and possibly alters an external state, in small-step semantics the program is part of the state that gets altered. With each step, the small-step interpreter mutates the term that represents the program. This process is destructive, as past states cannot be used to inform the behavior of a step, and neither can we use them in our proofs. As is conventional, it is precisely because of this that we introduced *intermediary terms* to the syntax. The following reduction scheme demonstrates this reasoning; it showcases the usage of two intermediary terms (`init` and reference literals) when creating a writeable reference.

$$\begin{aligned}
\texttt{new w\&Nat ((1 + 2) - 3)} &\xrightarrow{\text{step}} \\
\texttt{init w\&Nat 0x13ff1400 ((1 + 2) - 3)} &\xrightarrow{\text{step}} \\
\texttt{init w\&Nat 0x13ff1400 (3 - 3)} &\xrightarrow{\text{step}} \\
\texttt{init w\&Nat 0x13ff1400 0} &\xrightarrow{\text{step}} \\
\texttt{0x13ff1400 : w\&Nat} &
\end{aligned}$$

As shown, a `new` term reduces to an `init` term that eventually reduces to a reference literal. Syntactically, a reference literal is composed by an address and by a type; the address is the index to the referenced memory cell, and

the type is the type of references that point to that address. An `init` term is composed by these same components, plus a subterm that encapsulates the constructor of the reference.

We introduced reference literals to the language because we needed a term that represented the concept of a reference. In the previous example, the term $\langle$ `0x13ff1400 : w&Nat` $\rangle$ represents the writeable reference associated to the address `0x13ff1400`. Conversely, we introduced `init` terms to the language because we needed (for the sake of the proofs) to be able to syntactically ascertain whether or not a term was in the process of initializing a reference. Looking at the example, it is easy to assess that the writeable reference identified by the address `0x13ff1400` is being initialized during the second and fourth steps. In addition, the existence of the `init` term also allows us to check for some of the safety restrictions we mentioned earlier. In the example, the $\langle$ `(1 + 2) - 3` $\rangle$ subterm only starts to be reduced in the second step, after we transformed `new` into `init`. This means that the `new` term itself is *static*, since none of its subcomponents mutate throughout the reduction. Thus, to assess whether or not a term respects the boundary rules for monitored reference initialization, we must simply check for the presence of writeable references within `new` terms.

As to other intermediary terms, a `cr` term encapsulates a critical region much like an `init` term encapsulates a reference constructor. We use `cr` terms to define syntactically observable boundaries for critical regions during the reduction. They are the immediate result of reducing an `acq` term, analogous to how an `init` term is the immediate result of reducing a `new` term. The following scheme showcases this process:

$$\begin{aligned}
\texttt{acq (0x1a00ff21 : x\&Nat)}\,(\lambda\, i\,.\, i + 1) &\xrightarrow{\text{step}} \\
\texttt{cr 0x1a00ff21 (}\,42 + 1\,\texttt{)} &\xrightarrow{\text{step}} \\
\texttt{cr 0x1a00ff21}\ 43 &\xrightarrow{\text{step}} \\
43 &
\end{aligned}$$

The reduction step of an `acq` term acquires the lock of the associated monitored reference, reads its contents, and passes it as the argument to the function attached to the term. (It also substitutes instances of `self` inside the function for the value of the monitored reference.) In the example, the first step acquires the lock of the monitored reference `0x1a00ff21`, reads the value 42 from that address, and substitutes the variable i for 42 in the body of the function. It is the reduction of the `cr` term that actually evaluates the function to completion, as seen in the second step. In the last step, the `cr` term releases

σ	:=		SAFE TYPES
	|	`Unit`	*unit type*
	|	`Nat`	*numeric type*
	|	`r&`σ	*read-only reference type*
	|	`x&`τ	*monitored reference type*
τ	:=		TYPES
	|	σ	*safe type*
	|	`w&`τ	*writeable reference type*
	|	$\tau_1 \to \tau_2$	*function type*

Figure 3.2: Abstract syntax – types (inductive types `sty` and `ty` in `Sem.v`).

the lock and results in the value returned by the function. Syntactically, the address *ad* in the term $\langle$ `cr` *ad t* $\rangle$ refers to the associated monitored reference, while the subterm *t* refers to the term being evaluated inside the critical region.

In regard to waiting semantics, there are, as the list of terms shows, no `await` operations in the abstract syntax, only `wait` terms. We implement `await` operations by wrapping a `wait` with a loop, similar to what we have seen in Figure 2.4. This minimizes complexity, as `wait` is the simplest construct we need to model waiting semantics, and less complex constructs lead to less complex proofs.

$$\texttt{await}\ t \quad \equiv \quad \texttt{while}\ (\texttt{not}\ t)\ (\texttt{wait self})$$

As we can see, `wait` is not attached to a condition; its parameter simply indicates which monitored reference is associated to that `wait`. Remember that when an `acq` term gets reduced, it substitutes instances of `self` for the value of the acquired monitored reference. Thus, every $\langle$ `wait self` $\rangle$ from an `await` eventually becomes a $\langle$ `wait (`*ad* `:x&`τ`)` $\rangle$, which correctly identifies the monitored reference the `wait` pertains to. During evaluation, a `wait` term releases the lock of its monitored reference and reduces to a `reacq` intermediary term that, in turn, blocks while trying to reacquire the same lock. We implement `wait` semantics like this because `acq` terms are dynamic, and we cannot know at compile-time which address to use for a `wait`. We will explain the restrictions around waiting in more detail in the next chapter.

Figure 3.2 lists Elo's types as used by the abstract syntax. Note that the formalizations differentiate safe types from general types by construction. We represent safe types with the Greek letter $\langle \sigma \rangle$ and general types with the Greek letter $\langle \tau \rangle$. Safe types include `Unit`, `Nat`, read-only reference types, and monitored reference types, while general types include the safe types plus writeable references and function types. Moreover, it is worth highlighting the role of the inner types within each reference type: a read-only reference type

v	:=		VALUES
	|	`unit`	*unit value*
	|	n	*numeric value*
	|	$ad : \tau$	*address value*
	|	$\lambda\, a : \tau\, .\, t$	*function value*

Figure 3.3: Values (inductive type `value` in `Sem.v`).

contains an inner safe type (marked by the σ in $\langle$ `r&`σ $\rangle$), which enforces by construction that read-only references must be recursively safe; the monitored reference type is special because it is a safe type that can contain an unsafe type (as evidenced by the τ in $\langle$ `x&`τ $\rangle$); lastly, writeable reference types are standard, and can point to any other type.

3.1.1 Values

Figure 3.3 lists Elo's values, which, as we mentioned, are terms that cannot be further reduced. In the coming sections, we use $\langle\, v \,\rangle$ to indicate that a term is a value. We also use `value` as a predicate, as in $\langle$ `value` $t \rangle$, to the same effect.

As is common, we do not evaluate inside the body of function abstractions, so they are considered values.

3.2 Operational Semantics

We use small-step operational semantics to define Elo's evaluation. As we discussed earlier, in small-step semantics we define the execution of programs in terms of single-step reductions. The real semantics of the language is the transitive closure over the single-step relations; in short, we evaluate programs by performing zero or more steps. This method allows us to inductively reason about all the intermediary steps of an execution without needing to care whether it finishes or not.

Usually, small-step operational semantics are defined by a sole step relation that describes the behavior of the reductions. The semantics of Elo is a bit different. Instead of having one step relation, it has three hierarchical relations: concurrent step (C–Step), memory step (M–Step), and term step (T–Step). C–Step is the main reduction relation of the language, but we define it in conjunction with the other two relations. In essence, T–Step describes solely how a term reduces; M–Step describes how a term step interacts with the memory; and C–Step describes how a memory step interacts with the thread pool.

e	:=		Effects
	|	ALLOC $ad\ \tau$	*memory allocation*
	|	INIT $ad\ t$	*memory initialization*
	|	READ $ad\ t$	*memory read*
	|	WRITE $ad\ t$	*memory write*
	|	ACQ $ad\ t$	*acquire lock*
	|	REL ad	*release lock*
	|	WACQ ad	*acquire lock (wait)*
	|	WREL ad	*release lock (wait)*
	|	SPAWN t	*thread creation*
	|	NONE	*no effect*

Figure 3.4: Effects (inductive type `eff` in `Sem.v`).

This modular design was inspired by the multi-core architecture of computers: T–Step represents a core, M–Step represents the memory controller unit, and C–Step represents the thread pool controller inside the operating system. The main benefit of this hierarchical definition is that it allowed us to prove language invariants in a didactic progression, in the sense that we first proved properties on T–Step, then on M–Step, and finally on C–Step.

Elo's operational semantics is a *Labelled Transition System* [20]. In that sense, we use *effects* as the labels that represent the interactions between the three steps. Thus, effects pose as the bus in our analogy, since they allow the steps to "communicate" with each other. To illustrate, let us think of how dereferencing works. As mentioned, term steps do not involve the memory; still, the term step for dereferencing must somehow gain access to the value that was read from the memory. It *receives that value* through a READ effect. In turn, the dereferencing memory step (that does not know how terms reduce) reads the value from the memory and *sends it through* the READ effect to be used by the term step. We will describe this interaction in more details in the coming sections.

As seen in Figure 3.4, there are ten types of *effects*. The ALLOC, INIT, READ, and WRITE effects apply to typical memory operations; ACQ and REL apply to `acquire` synchronization; WACQ and WREL apply to `wait` synchronization; SPAWN applies to thread creation; and NONE indicates the absence of an effect. The NONE effect is used by steps that neither interact with the memory nor the thread pool. This applies, for example, to when a step adds two numbers or calls a function.

It may seem that some effects in the list are redundant. In particular, we have two effects for acquiring locks (ACQ and WACQ) and two effects for releasing locks (REL and WREL). This differentiation, however, is vital to the inner workings of our proofs, since some of our lemmas and theorems depend

on us knowing whether we are using an `acquire` or a `wait` to acquire/release a lock. We also have separate effects for allocation and initialization (ALLOC and INIT), because, as we will see, the operational semantics executes allocation and initialization in separate steps.

Let us now detail the formal semantics of each step relation.

3.2.1 Term step

T–Step describes how a thread evolves, abstracting away details of memory operations and thread creation. Figure 3.5 lists all the reduction rules for the T–Step relation. The notation $\langle\, t_1 \xrightarrow{e} t_2 \,\rangle$ states that the term t_1 reduces to the term t_2 with the effect e. In the Coq code, we implement T–Step using the inductive type `tstep` (`Sem.v`).

Small-step reduction rules for terms fall into one of two categories: they can either be *computation* rules or *congruence* rules. Computation rules describe the essence of how to reduce a term, while congruence rules describe how to recursively apply a computation rule to the subterm of a term.

In the interest of conciseness, we chose to present the congruence rules for T–Step using *evaluation contexts*. Together with the $\textsc{T}_{\texttt{ctx}}$ rule, an evaluation context (ε) serves as a compact representation of a congruence rule. A *hole* $\langle\, \square \,\rangle$ in an evaluation context indicates which subterm is being reduced by the associated congruence rule. For example, the evaluation contexts $\langle\, \square \texttt{ + } t \,\rangle$ and $\langle\, v \texttt{ + } \square \,\rangle$ condense, respectively, the PLUS1 and PLUS2 congruence rules for addition:

$$\textsc{plus}_1 \frac{t_1 \xrightarrow{e} t_2}{(\, t_1 \texttt{ + } t \,) \xrightarrow{e} (\, t_2 \texttt{ + } t \,)} \qquad \textsc{plus}_2 \frac{t_1 \xrightarrow{e} t_2}{(\, v \texttt{ + } t_1 \,) \xrightarrow{e} (\, v \texttt{ + } t_2 \,)}$$

The PLUS1 rule states that, if t_1 reduces to t_2 with an effect e then $\langle\, t_1 + t \,\rangle$ reduces to $\langle\, t_2 + t \,\rangle$ with that same effect. The PLUS2 rule states that if t_1 reduces to t_2 with an effect e then $\langle\, v + t_1 \,\rangle$ reduces to $\langle\, v + t_2 \,\rangle$ with that same effect; we use v in the first subterm to enforce that it must be a value. Together, PLUS1 and PLUS2 formalize how to reduce the subterms of an addition. The "translation" for the other evaluation contexts follow the same logic.

Note that we do not have any congruence rules for the `new` term. This demonstrates that the subterm t in $\langle\, \texttt{new } \tau \ t \,\rangle$ is static. After we reduce `new` into `init`, then its subterm reduces as evidenced by the $\langle\, \texttt{init } \tau \ ad \ \square \,\rangle$ evaluation context. Similarly, the subterm t_2 in a term $\langle\, \texttt{acq } t_1 \,(\lambda a \,.\, t_2) \,\rangle$ is also static. After we reduce `acq` into `cr`, then its subterm reduces as evidenced by the $\langle\, \texttt{cr } ad \ \square \,\rangle$ evaluation context.

$$
\begin{array}{lrcll}
\mathrm{T}_{\texttt{plus}} & n_1 \texttt{ + } n_2 & \xrightarrow{\text{NONE}} & n_1 + n_2 & \\
\mathrm{T}_{\texttt{monus}} & n_1 \texttt{ - } n_2 & \xrightarrow{\text{NONE}} & n_1 - n_2 & \\
\mathrm{T}_{\texttt{seq}} & v\texttt{; } t & \xrightarrow{\text{NONE}} & t & \\
\mathrm{T}_{\texttt{if1}} & \texttt{if 0 } t_1\ t_2 & \xrightarrow{\text{NONE}} & t_2 & \\
\mathrm{T}_{\texttt{if2}} & \texttt{if } n\ t_1\ t_2 & \xrightarrow{\text{NONE}} & t_1 & (\text{when } n \neq 0) \\
\mathrm{T}_{\texttt{while}} & \texttt{while } t_1\ t_2 & \xrightarrow{\text{NONE}} & \texttt{if } t_1\ \texttt{(}t_2\texttt{; while } t_1\ t_2\texttt{) unit} & \\
\mathrm{T}_{\texttt{call}} & \texttt{(}\lambda a : \tau . t\texttt{) } v & \xrightarrow{\text{NONE}} & [a/v]t & \\
\mathrm{T}_{\texttt{new}} & \texttt{new } \tau\ t & \xrightarrow{\text{ALLOC } ad\ \tau} & \texttt{init } \tau\ ad\ t & \\
\mathrm{T}_{\texttt{init}} & \texttt{init } \tau\ ad\ v & \xrightarrow{\text{INIT } ad\ v} & ad : \tau & \\
\mathrm{T}_{\texttt{load}} & \texttt{!(}ad : \tau\texttt{)} & \xrightarrow{\text{READ } ad\ v} & v & \\
\mathrm{T}_{\texttt{asg}} & \texttt{(}ad : \tau\texttt{) := } v & \xrightarrow{\text{WRITE } ad\ v} & \texttt{unit} & \\
\mathrm{T}_{\texttt{acq}} & \texttt{acq (}ad : \tau\texttt{)(}\lambda a\texttt{ . }t\texttt{)} & \xrightarrow{\text{ACQ } ad\ v} & \texttt{cr } ad\ \texttt{([self/(}ad : \tau\texttt{)][}a/v\texttt{]}t\texttt{)} & \\
\mathrm{T}_{\texttt{cr}} & \texttt{cr } ad\ v & \xrightarrow{\text{REL } ad} & v & \\
\mathrm{T}_{\texttt{wait}} & \texttt{wait (}ad : \tau\texttt{)} & \xrightarrow{\text{WREL } ad} & \texttt{reacq } ad & \\
\mathrm{T}_{\texttt{reacq}} & \texttt{reacq } ad & \xrightarrow{\text{WACQ } ad} & \texttt{unit} & \\
\mathrm{T}_{\texttt{spawn}} & \texttt{spawn } t & \xrightarrow{\text{SPAWN } t} & \texttt{unit} & \\
\mathrm{T}_{\texttt{ctx}} & \varepsilon\{t_1\} & \xrightarrow{e} & \varepsilon\{t_2\} & (\text{if } t_1 \xrightarrow{e} t_2)
\end{array}
$$

$$
\begin{array}{rcllll}
\varepsilon & := & \square \texttt{ + } t & \mid v \texttt{ + } \square & \mid \square \texttt{ - } t & \mid v \texttt{ - } \square \\
& \mid & \square\texttt{; } t & \mid \texttt{if } \square\ t_1\ t_2 & \mid \texttt{while } \square\ t & \\
& \mid & \square\ t & \mid v\ \square & & \\
& \mid & \texttt{init } \tau\ ad\ \square & \mid \texttt{!}\square & \mid \square \texttt{ := } t & \mid v \texttt{ := } \square \\
& \mid & \texttt{acq } \square\ \texttt{(}\lambda a\texttt{ . }t\texttt{)} & \mid \texttt{cr } ad\ \square & &
\end{array}
$$

Figure 3.5: T–Step (inductive type `tstep` in `Sem.v`).

As to the computation rules, we can see that the rules for plus, monus, sequencing, conditionals, loops, and function calls do not produce effects, their semantics follow the standard for these constructions: T`plus` adds two numbers; T`monus` subtracts two numbers; T`seq` reduces to the second subterm when the first subterm is a value; T`if` reduces to the second or third subterm, depending on whether or not the first subterm is zero; T`while` unfolds to an `if` term containing a copy of the original `while`; and T`call` performs variable substitution, represented as $\langle [x / v] t \rangle$, which reads "substitute instances of the variable x for the value v inside the term t".

The other rules are more interesting, since they depend on effects. For instance, if we analyze T`load` disregarding the READ effect, it basically "creates" a value. Since T–Step has no notion of a memory, the origin of that value is not obvious. In a sense, we can interpret the READ effect as consuming an address ad from the term $\langle \texttt{!(}ad : \tau\texttt{)} \rangle$ and producing a value v. Locally, T`load` trusts, in an oracle fashion, that the value it received from the effect is the value in the memory cell indexed by the sent address. As we will see later, there will be a memory step paired with T`load` that defines the memory semantics for dereferencing and that guarantees the validity of the value.

(It is worth noting that, even though we used produce/consume idioms to explain effects, logic relations do not have directionality. For example, we read the $\rightarrow$ arrow as a reduction from left to right, but we are actually defining an association between elements in a set. Similarly, despite our interpretation that ad is being sent and v is coming from the READ effect, T`load` is, in logical terms, merely a relation between an address, a type, and a value.)

The same principles behind T`load` apply to the other rules:

- The ALLOC effect in T`new` consumes the type from the term $\langle \texttt{new}\ \tau\ t \rangle$, and produces an address ad for the term $\langle \texttt{init}\ \tau\ ad\ t \rangle$. The type informs the paired memory step which kind of reference cell to allocate, and the returned address is the index to that memory cell.
- The INIT effect in T`init` consumes the address and the value from the term $\langle \texttt{init}\ \tau\ ad\ v \rangle$, and the step reduces to a reference literal $\langle ad : \tau \rangle$. Unlike ALLOC and READ, INIT is a one-way effect that does not produce values or addresses to be used in the reduction. The paired memory step takes care of initializing the memory cell associated with the given address using the provided value.
- The WRITE effect in T`asg` consumes the address and the value from the term $\langle \texttt{(}ad : \tau\texttt{)}\ \texttt{:=}\ v \rangle$, and the step reduces to `unit`. Overall, the WRITE effect behaves much like INIT.

- The ACQ effect in the $\text{T}_{\texttt{acq}}$ rule consumes the address ad from the term $\langle\, \texttt{acq (}\, ad : \tau \,\texttt{)(}\lambda\, a \,.\, t\texttt{)} \,\rangle$, and produces a value v for the resulting term $\langle\, \texttt{cr}\ ad\ \texttt{(}[\,\texttt{self}\,/\,\texttt{(}\, ad : \tau \,\texttt{)}\,][\,a\,/\,v\,]\,t\,\texttt{)} \,\rangle$. The $\text{T}_{\texttt{acq}}$ rule is like a dereferencing operation mixed with a function call. It sends to the effect the address of the monitored reference, and it receives a value in return. We expect the value to be the contents of the monitored reference, and we use it as the value of the parameter a when calling the function attached to the `acquire`. The function call is represented by the substitution $\langle\, [\,a\,/\,v\,]\,t \,\rangle$, and, on top of that, we also substitute any instances of `self` with the value of the monitored reference. The paired memory step ensures the synchronization semantics responsible for acquiring the lock.
- The REL effect in $\text{T}_{\texttt{cr}}$ consumes the address from the term $\langle\, \texttt{cr}\ ad\ v \,\rangle$, and the step reduces to the value v (which is the value returned by the function called by the `acquire`). Like with the ACQ effect, the paired memory step ensures the synchronization semantics responsible for releasing the lock.
- The WREL effect in $\text{T}_{\texttt{wait}}$ consumes the address ad from $\langle\, \texttt{wait (}\, ad : \tau \,\texttt{)} \,\rangle$, and the step reduces to the term $\langle\, \texttt{reacq}\ ad \,\rangle$. In turn, the WACQ effect in $\text{T}_{\texttt{reacq}}$ consumes the address ad from the term $\langle\, \texttt{reacq}\ ad \,\rangle$, and the step reduces to `unit`. In the eyes of the T–Step relation, the $\text{T}_{\texttt{wait}}$ and $\text{T}_{\texttt{reacq}}$ rules are quite straightforward. The memory steps paired with $\text{T}_{\texttt{wait}}$ and $\text{T}_{\texttt{reacq}}$ perform the relevant synchronization operations using the given address ad.
- The SPAWN effect in $\text{T}_{\texttt{spawn}}$ consumes the term t from $\langle\, \texttt{spawn}\ t \,\rangle$ to create a new thread, and the step reduces to `unit`. Unlike the other effects that pair with rules from M–Step, we will see that $\text{T}_{\texttt{spawn}}$ pairs with a rule from C–Step, because it needs to communicate with the thread pool instead of the memory.

3.2.2 Memory step

Before we describe the M–Step relation, let us first discuss the formalization for the memory. We use a list to represent the memory in Elo. Addresses correspond to indices in this list. Each entry in the list represents a memory cell, which contains a term, a boolean, a type, and an owner. We use the notation $[\![\, t\,,\ \chi\,,\ \tau\,,\ \omega \,]\!]$ to represent memory cells containing those four elements. The term t is the de facto value being stored in that memory cell. Cells that were allocated but not initialized hold the special empty value `none` in place

of a term. The boolean χ indicates whether or not that memory cell is locked. All memory cells posses this field, even those that have nothing to do with monitored references. (We will discuss this suboptimal allocation of memory later.) The type τ is a reference type (also called a *pointer type*), such that the its internal type is the type of the term that is stored in the cell. Finally, the owner ω identifies either a thread or a monitored reference, indicating who can access the memory cell.

Only the first two elements in a memory cell (the term and the lock) are relevant to the step relations. The operational semantics sets the other two (the type and the owner), but does not use their values in the reductions. They exist in the memory cell solely for the sake of our proofs. In an actual Elo implementation, there is no need for types and owners in the memory cells.

Regarding list operations, there are five basic operations that deal with retrieving information from the memory representation:

- $\#m$ (*length*): returns the length of the list that represents the memory. This operation is specially important because it indicates the address of the last allocated memory cell.
- $m[ad].t$ (*get term*): returns the term stored in the cell (or `none` in case of an uninitialized or invalid address).
- $m[ad].\chi$ (*get lock*): returns a boolean that indicates whether or not the cell is locked (or `false` in case of an invalid address).
- $m[ad].\tau$ (*get type*): returns the pointer type of the cell (or `Unit` in case of an invalid address).
- $m[ad].\omega$ (*get owner*): returns the owner of the cell (or the invalid owner `invalid` in case of an invalid address).

Additionally, there are five operations that mutate the memory. All of them return the changed memory.

- $m[ad \xleftarrow{\texttt{term}} t]$ (*set term*): sets the term stored in the cell indexed by the given address (or does nothing in case of an invalid address). We use this operation when assigning to or initializing a reference.
- $m[ad \xleftarrow{\texttt{lock}} \texttt{true/false}]$ (*set lock*): locks or unlocks the cell (or does nothing in case of an invalid address).
- $m[ad \xleftarrow{\texttt{owner}} \omega]$ (*set owner*): sets the owner of the cell (or does nothing in case of an invalid address). We use this operation in the C–Step relation instead of M–Step. We will explain its purpose then.

- $m \mathbin{+\!\!+\!\!+} \tau$ (*add cell*): adds the cell $[\![\texttt{none}, \texttt{false}, \tau, \texttt{invalid}]\!]$ to the end of the memory. We use this operation when allocating new memory cells. Note that a new cell does not contain a term, it is unlocked, it has no valid owner, but it does have a type. (This operation is implemented using the $\langle \mathbin{+\!\!+} \rangle$ operation that appends two lists.)

We implemented these list operations in Coq in the `Array.v` and `Sem.v` files.

Figure 3.6 lists the reduction rules for the M–Step relation. The notation $\langle\, m_1 \,/\, t_1 \overset{e}{\Rightarrow} m_2 \,/\, t_2 \,\rangle$ indicates that the memory m_1 and term t_1 reduce to the memory m_2 and the term t_2 with the effect e.

In essence, M–Step expands the semantics of T–Step by also describing how the memory evolves with each reduction. To clarify this notion, let us analyze the $\textsc{M}_{\texttt{write}}$ rule. The term step $\langle\, t_1 \xrightarrow{\textsc{write}\ ad\ t} t_2 \,\rangle$ in the premise of the rule determines how the initial term t_1 reduces to t_2. While the term step is ignorant to the purpose of the address and the term sent through the WRITE effect, the memory step uses the received address and term to reduce the original memory m to $m[ad \xleftarrow{\texttt{term}} t]$, effectively mutating it. Thus, $\textsc{M}_{\texttt{write}}$ expands the (term) semantics of $\textsc{T}_{\texttt{asg}}$ to include memory semantics.

As another example, let us analyze the $\textsc{M}_{\texttt{read}}$ rule. Again, the T–Step in the premise determines how the term reduces. However, the $\textsc{M}_{\texttt{read}}$ rule, as the name implies, does not alter the memory; it simply receives an address from the READ effect, and sends the term read from that address back to the effect . Different from $\textsc{M}_{\texttt{write}}$, the $\textsc{M}_{\texttt{read}}$ rule uses the READ effect as a "two-way" channel. This may seen strange given the perceived direction of the step relations implied by the arrow notation, but remember that these are *logical relations* that do not abide to this notion of directionality. Despite the intuition that we "send to" and "receive from" an effect, their true purpose is to add constraints to both the term and the memory steps.

More generally, we can see that T–Step reductions appear as premises in all the rules for M–Step, each with a different effect. The effects filter which T–Step rules are valid for each M–Step rule, and establish the pairing between T–Step and M–Step we mentioned in the previous section. Besides that, it is also obvious that T–Step encapsulates the semantics of term reduction, in such a way that M–Step itself does not need to know anything about it. With all that in mind, the remaining inference rules are easy enough to understand:

- $\textsc{M}_{\texttt{none}}$ is a pass-through rule that does not alter the memory. It allows M–Step to encompass all the term–step reductions that produce no effects.

$$\mathrm{M}_{\texttt{none}}\ \frac{t_1 \xrightarrow{\text{NONE}} t_2}{m \,/\, t_1 \xRightarrow{\text{NONE}} m \,/\, t_2}$$

$$\mathrm{M}_{\texttt{alloc}}\ \frac{t_1 \xrightarrow{\text{ALLOC } \#m\ \tau} t_2}{m \,/\, t_1 \xRightarrow{\text{ALLOC } \#m\ \tau} (m \mathbin{+\!\!+\!\!+} \tau) \,/\, t_2}$$

$$\mathrm{M}_{\texttt{init}}\ \frac{t_1 \xrightarrow{\text{INIT } ad\ t} t_2}{m \,/\, t_1 \xRightarrow{\text{INIT } ad\ t} m[ad \xleftarrow{\texttt{term}} t] \,/\, t_2}$$

$$\mathrm{M}_{\texttt{read}}\ \frac{m[ad].t = t_e \qquad t_1 \xrightarrow{\text{READ } ad\ t_e} t_2}{m \,/\, t_1 \xRightarrow{\text{READ } ad\ t_e} m \,/\, t_2}$$

$$\mathrm{M}_{\texttt{write}}\ \frac{t_1 \xrightarrow{\text{WRITE } ad\ t} t_2}{m \,/\, t_1 \xRightarrow{\text{WRITE } ad\ t} m[ad \xleftarrow{\texttt{term}} t] \,/\, t_2}$$

$$\mathrm{M}_{\texttt{acq}}\ \frac{m[ad].\mathcal{X} = \texttt{false} \qquad m[ad].t = t_e \qquad t_1 \xrightarrow{\text{ACQ } ad\ t_e} t_2}{m \,/\, t_1 \xRightarrow{\text{ACQ } ad\ t_e} m[ad \xleftarrow{\texttt{lock}} \texttt{true}] \,/\, t_2}$$

$$\mathrm{M}_{\texttt{rel}}\ \frac{t_1 \xrightarrow{\text{REL } ad} t_2}{m \,/\, t_1 \xRightarrow{\text{REL } ad} m[ad \xleftarrow{\texttt{lock}} \texttt{false}] \,/\, t_2}$$

$$\mathrm{M}_{\texttt{wacq}}\ \frac{m[ad].\mathcal{X} = \texttt{false} \qquad t_1 \xrightarrow{\text{WACQ } ad} t_2}{m \,/\, t_1 \xRightarrow{\text{WACQ } ad} m[ad \xleftarrow{\texttt{lock}} \texttt{true}] \,/\, t_2}$$

$$\mathrm{M}_{\texttt{wrel}}\ \frac{t_1 \xrightarrow{\text{WREL } ad} t_2}{m \,/\, t_1 \xRightarrow{\text{WREL } ad} m[ad \xleftarrow{\texttt{lock}} \texttt{false}] \,/\, t_2}$$

Figure 3.6: M–Step (inductive type `mstep` in `Sem.v`).

- $\text{M}_{\texttt{alloc}}$ expands the memory, allocating at its end a new cell with the type it received from $\text{T}_{\texttt{new}}$ through the ALLOC effect. It sends back the address $\langle\, \#m \,\rangle$ to the term step, which references the newly inserted cell.
- $\text{M}_{\texttt{init}}$ initializes a memory cell. It receives an address and a term from the INIT effect, and uses the *set term* operation to mutate the memory. It functions exactly like $\text{M}_{\texttt{write}}$, but using the INIT effect instead of the WRITE effect.
- $\text{M}_{\texttt{acq}}$ locks an unlocked memory cell and reads its contents. It receives the relevant address from the ACQ effect, and sends back the read term. Note that a thread cannot take this step while the cell is locked (while $m[ad].\chi = \texttt{true}$), which corresponds to the usual blocking semantics of locks.
- $\text{M}_{\texttt{rel}}$ unlocks the memory cell referenced by the address it received from the REL effect. (It does not care whether or not the cell was locked in the first place.)
- $\text{M}_{\texttt{wacq}}$ locks an unlocked memory cell. It receives the relevant address from the WACQ effect, and (like $\text{M}_{\texttt{acq}}$) it replicates the blocking semantics of locks.
- $\text{M}_{\texttt{wrel}}$ functions exactly like $\text{M}_{\texttt{rel}}$, but using the WREL effect instead of REL.

3.2.3 Concurrent step

The C–Step relation is the main step relation of the language. It expands on T–Step and M–Step to add the last missing piece of the semantics: multithreading. To that end, we represent the thread pool in Elo as a list of terms. We use the operation $ths[tid]$ to retrieve a thread from the list ths given the thread identifier tid, and the operation $ths[tid \leftarrow t]$ to update a thread in the list with a new term t. These two operations are important because they manipulate threads individually, which, as we will see, is how C–Step handles them. We also use $\langle\, ths \mathbin{+\!\!+} \texttt{[}t\texttt{]} \,\rangle$ to add a new thread to the thread pool, and $\#ths$ to get the length of the list of threads.

The C–Step relation detailed in Figure 3.7 describes how a memory and a thread pool evolve in one step. The notation $\langle\, m_1 \,/\, ths_1 \overset{(tid,\ e)}{\rightsquigarrow\!\!\triangleright} m_2 \,/\, ths_2 \,\rangle$ indicates that the memory m_1 and the thread pool ths_1 reduce to the memory m_2 and the thread pool ths_2. Despite the parallel nature of processing multiple thread reductions, the relation itself only performs one reduction at one thread

$$\textsc{C}_{\texttt{mem}}\ \frac{e \neq \textsc{alloc} \quad tid < \#ths \quad m_1 \,/\, ths[tid] \overset{e}{\Rightarrow} m_2 \,/\, t}{m_1 \,/\, ths \overset{(tid,\ e)}{\rightsquigarrow} m_2 \,/\, ths[tid \leftarrow t]}$$

$$\textsc{C}_{\texttt{alloc}}\ \frac{tid < \#ths \quad \omega = \texttt{gcr}\ ths[tid]\ tid \quad m_1 \,/\, ths[tid] \xRightarrow{\textsc{alloc}\ ad\ \tau} m_2 \,/\, t}{m_1 \,/\, ths \overset{(tid,\ \textsc{alloc}\ ad\ \tau)}{\rightsquigarrow} m_2[ad \xleftarrow{\texttt{owner}} \omega] \,/\, ths[tid \leftarrow t]}$$

$$\textsc{C}_{\texttt{spawn}}\ \frac{tid < \#ths \quad ths[tid] \xrightarrow{\textsc{spawn}\ t'} t}{m \,/\, ths \overset{(tid,\ \textsc{spawn}\ t)}{\rightsquigarrow} m \,/\, (ths[tid \leftarrow t] +\!\!+ [t'])}$$

Figure 3.7: C–Step (inductive type `cstep` in `Sem.v`).

at a time. In the notation, the thread identifier besides the effect indicates which thread was stepped.

There are only three rules for C–Step. All of them check whether the ID of the thread performing the step is within the bounds of the thread pool.

- The Cmem rule encompasses the semantics of all M–Step rules except for Malloc (since it filters out the ALLOC effect). The M–Step rule in the premise defines how the memory and the thread reduce, while Cmem simply updates the value of the stepped thread in the thread pool.
- The Calloc rule mimics the Cmem rule in the context of the ALLOC effect, while also inserting the owner of the allocated address in the memory. If the address pertains to a read-only or monitored reference, than the stored owner for that address is irrelevant, since we are only interested in keeping track of writeable references. The Calloc rule uses the `gcr` (*get current region*) function to compute the owner. The function checks for $\langle$ `cr` *ad t* $\rangle$ and $\langle$ `init x&`τ *ad t* $\rangle$ subterms in the thread, and returns the innermost address *ad* it finds. This address identifies the owner of the cell allocated by the step. In case `gcr` does not find any `cr` or `init` terms, then the owner of the allocated cell is the thread itself. (This is why we could not compute the owner in the Malloc rule; we needed the thread identifier.) Remember that we do not use this owner information in the semantics, only in the context of the proofs.
- The Cspawn rule handles thread creation. It adds the term it received through the SPAWN effect to the thread pool.

C–Step is specially interesting because, at first glance, it does not seem to be defining concurrent behavior. However, if we look closer, we can see that none of its rules constrain on the order in which the threads should be stepped.

$$\mathrm{U}_{\texttt{first}}\ \frac{}{m \,/\, ths \overset{\{\}}{\rightsquigarrow\!\triangleright}{}^{*}\ m \,/\, ths}$$

$$\mathrm{U}_{\texttt{trans}}\ \frac{\begin{array}{l} m_1 \,/\, ths_1 \overset{tc}{\rightsquigarrow\!\triangleright}{}^{*}\ m_2 \,/\, ths_2 \\ m_2 \,/\, ths_2 \overset{(tid,\ e)}{\rightsquigarrow\!\triangleright}\ m_3 \,/\, ths_3 \end{array}}{m_1 \,/\, ths_1 \overset{(tid,\ e)\,::\,tc}{\rightsquigarrow\!\triangleright}{}^{*}\ m_3 \,/\, ths_3}$$

Figure 3.8: Multistep (inductive type `multistep` in `Sem.v`).

Thus, it allows for every possible non-deterministic interleaving between thread reductions, which ensures the intended semantics.

3.2.4 Multistep

Figure 3.8 presents Elo's multistep reduction relation. It steps a program (a memory and a thread pool) zero or more times and produces a trace. A trace is a list of events, with each event containing a thread identifier and an effect; they serve as points in the discrete timeline of that program's execution. The trace allows us to analyze the order in which the threads were stepped and the effects they produced. As we will see, traces are vital to the structure of our proofs.

Regarding the rules, $\mathrm{U}_{\texttt{first}}$ starts the reduction with an empty trace represented by `{}`, while $\mathrm{U}_{\texttt{trans}}$ is the transitive rule that appends one more step to a previous multistep computation. Note that we use the *cons* operator $\langle :: \rangle$ to prepend events to the trace list, instead of appending with $\langle +\!\!+ \rangle$ like we did with the memory and the thread pool. We do this for convenience, since it simplifies some of the proofs when we apply the induction principle. As a consequence, the events in a trace appear in reverse: the first event in the list was the last event to take place.

3.3 Typing

Elo's typing rules define a typing relation that relates terms to their types. A typing judgment $\langle \Gamma \vdash t \in \tau \rangle$ states that, given the typing environment Γ, the term t has type τ. A typing environment is a mapping from variables to their types; we use them to model scope rules for variables. Regarding notations for maps, a $\varnothing$ represents the empty map, $\Gamma\,[\,x\,]$ represents the lookup operation, and $\Gamma\,[\,x \Leftarrow \tau\,]$ represents the update operation. Lookup returns a special value $\bot$ if a variable does not have a type (if a variable is not present in the map).

$$\tau_{\texttt{unit}}\frac{}{\Gamma \vdash \texttt{unit} \in \texttt{Unit}} \qquad \tau_{\texttt{nat}}\frac{}{\Gamma \vdash n \in \texttt{Nat}}$$

$$\tau_{\texttt{plus}}\frac{\Gamma \vdash t_1 \in \texttt{Nat} \quad \Gamma \vdash t_2 \in \texttt{Nat}}{\Gamma \vdash t_1 \texttt{ + } t_2 \in \texttt{Nat}} \qquad \tau_{\texttt{monus}}\frac{\Gamma \vdash t_1 \in \texttt{Nat} \quad \Gamma \vdash t_2 \in \texttt{Nat}}{\Gamma \vdash t_1 \texttt{ - } t_2 \in \texttt{Nat}}$$

$$\tau_{\texttt{if}}\frac{\Gamma \vdash t_1 \in \texttt{Nat} \quad \Gamma \vdash t_2 \in \tau \quad \Gamma \vdash t_3 \in \tau}{\Gamma \vdash \texttt{if } t_1\ t_2\ t_3 \in \tau} \qquad \tau_{\texttt{while}}\frac{\Gamma \vdash t_1 \in \texttt{Nat} \quad \Gamma \vdash t_2 \in \tau}{\Gamma \vdash \texttt{while } t_1\ t_2 \in \texttt{Unit}}$$

$$\tau_{\texttt{seq}}\frac{\Gamma \vdash t_1 \in \tau_1 \quad \Gamma \vdash t_2 \in \tau_2}{\Gamma \vdash t_1\texttt{;}\ t_2 \in \tau_2} \qquad \tau_{\texttt{var}}\frac{\Gamma\,[\,x\,] = \tau}{\Gamma \vdash x \in \tau}$$

$$\tau_{\texttt{fun}}\frac{\Gamma\,[\,a \Leftarrow \tau_a\,] \vdash t \in \tau}{\Gamma \vdash \lambda\, a : \tau_a\,.\, t \in \tau_a \to \tau} \qquad \tau_{\texttt{call}}\frac{\Gamma \vdash t_1 \in \tau_a \to \tau \quad \Gamma \vdash t_2 \in \tau_a}{\Gamma \vdash t_1\ t_2 \in \tau}$$

Figure 3.9: Typing rules for the standard features (inductive type `type_of` in `Sem.v`).

In Coq, we implement the typing relation using the inductive type `type_of` (`Sem.v`). We define the map data structure in the `Map.v` file.

We present typing rules as a set of inference rules. Let us analyze some of the basic rules in Figure 3.9 before we delve into the more complex ones. The rules for `unit` ($\tau_{\texttt{unit}}$) and numbers ($\tau_{\texttt{nat}}$) are trivial; the language requires no premises (no conditions) to asses that `unit` has type `Unit` and that a number has type `Nat`. The typing rule for addition ($\tau_{\texttt{plus}}$) is also quite straightforward: both subterms of the addition must have numeric types, and then the addition itself has a numeric type as well. There is no scope or variable handling involved when adding terms, so the usage of typing environments is trivial. The typing rules for monus, conditionals, loops, sequencing, variables, and functions are standard and follow similar ideas, so we will refrain from discussing them.

The main non-conventional characteristic of the type system is the use of *safe environments*, which restrict variable visibility in the boundaries between threads and between a thread and a critical region. The typing rules for terms that define these boundaries make use of SAFE, an operation over typing environments that alters the result of the lookup operation by filtering out variables with non-safe types. We formalize SAFE as follows.

$$\textsc{safe}\,(\,\Gamma\,)\,[\,x\,] = \begin{cases} \Gamma\,[\,x\,] & \text{if } \Gamma\,[\,x\,] \in \sigma \\ \bot & \text{otherwise} \end{cases}$$

We use SAFE when typing monitored reference initialization, `acq` terms,

$$\tau_{\texttt{refR}}\frac{}{\Gamma \vdash ad : \texttt{r\&}\sigma \in \texttt{r\&}\sigma} \qquad \tau_{\texttt{refX}}\frac{}{\Gamma \vdash ad : \texttt{x\&}\tau \in \texttt{x\&}\tau} \qquad \tau_{\texttt{refW}}\frac{}{\Gamma \vdash ad : \texttt{w\&}\tau \in \texttt{w\&}\tau}$$

$$\tau_{\texttt{newR}}\frac{\Gamma \vdash t \in \sigma}{\Gamma \vdash \texttt{new r\&}\sigma\ t \in \texttt{r\&}\sigma} \qquad \tau_{\texttt{initR}}\frac{\varnothing \vdash t \in \sigma}{\Gamma \vdash \texttt{init r\&}\sigma\ ad\ t \in \texttt{r\&}\sigma}$$

$$\tau_{\texttt{newX}}\frac{\textsc{safe}(\Gamma) \vdash t \in \tau}{\Gamma \vdash \texttt{new x\&}\tau\ t \in \texttt{x\&}\tau} \qquad \tau_{\texttt{initX}}\frac{\varnothing \vdash t \in \tau}{\Gamma \vdash \texttt{init x\&}\tau\ ad\ t \in \texttt{x\&}\tau}$$

$$\tau_{\texttt{newW}}\frac{\Gamma \vdash t \in \tau}{\Gamma \vdash \texttt{new w\&}\tau\ t \in \texttt{w\&}\tau} \qquad \tau_{\texttt{initW}}\frac{\varnothing \vdash t \in \tau}{\Gamma \vdash \texttt{init w\&}\tau\ ad\ t \in \texttt{w\&}\tau}$$

$$\tau_{\texttt{loadR}}\frac{\Gamma \vdash t \in \texttt{r\&}\sigma}{\Gamma \vdash \texttt{!}t \in \sigma} \qquad \tau_{\texttt{loadW}}\frac{\Gamma \vdash t \in \texttt{w\&}\tau}{\Gamma \vdash \texttt{!}t \in \tau} \qquad \tau_{\texttt{asg}}\frac{\Gamma \vdash t_1 \in \texttt{w\&}\tau \quad \Gamma \vdash t_2 \in \tau}{\Gamma \vdash t_1 \texttt{ := } t_2 \in \texttt{Unit}}$$

Figure 3.10: Typing rules for memory operations (inductive type `type_of` in `Sem.v`).

and `spawn` terms, which enforces the safety guarantees discussed in section 2.2. For instance, the example in Figure 2.5 does not type check because SAFE purges the variable `a` (which has an unsafe type) from the typing environment of the `spawn` term.

The typing rules in Figure 3.10 apply to the basic memory operations. The first three rules establish the typing semantics for references. The suffixes in the names of the rules hint to which kind of reference each rule pertains to: `R` stands for read-only references, `X` stands for for monitored references, and `W` stands for writeable references. These rules may look trivial, but they perform the important task of checking that the type τ inside a term $\langle\, ad : \tau \,\rangle$ is always a reference type, which prevents, for example, terms like $\langle\, ad : \texttt{Nat} \,\rangle$ from type checking.

The rules for `new` ($\tau_{\texttt{newR}}$, $\tau_{\texttt{newX}}$, and $\tau_{\texttt{newW}}$) and `init` ($\tau_{\texttt{initR}}$, $\tau_{\texttt{initX}}$, and $\tau_{\texttt{initW}}$) mirror the logic of the reference rules. Note that, when typing `new` for monitored references, a `new` term is well-typed if its body is well-typed within the SAFE version of its parent's environment, which prevents unsafe values from leaking into a monitored reference through initialization. The same check is naturally not present when typing the allocation of a read-only or writeable reference. Moreover, we use the empty environment for `init` subterms because they cannot appear inside the body of an `acquire` or of a function abstraction, which are the terms that add variables to an environment. All we will see, this is also valid for the `cr` intermediary term.

The two rules for dereferencing ($\tau_{\texttt{loadR}}$ and $\tau_{\texttt{loadW}}$) concern read-only and writeable references, which enforces that we cannot read the contents

$$\tau_{\texttt{acq}}\frac{\Gamma \vdash t_1 \in \texttt{x\&}\tau \quad \textsc{safe}(\Gamma)[\texttt{self} \Leftarrow \texttt{x\&}\tau][a \Leftarrow \tau] \vdash t_2 \in \sigma}{\Gamma \vdash \texttt{acq}\ t_1\ (\lambda a\,.\,t_2) \in \sigma}$$

$$\tau_{\texttt{cr}}\frac{\varnothing \vdash t \in \tau}{\Gamma \vdash \texttt{cr}\ ad\ t \in \tau} \qquad \tau_{\texttt{spawn}}\frac{\textsc{safe}(\Gamma) \vdash t \in \tau}{\Gamma \vdash \texttt{spawn}\ t \in \texttt{Unit}}$$

$$\tau_{\texttt{wait}}\frac{\Gamma \vdash t \in \texttt{x\&}\tau}{\Gamma \vdash \texttt{wait}\ t \in \texttt{Unit}} \qquad \tau_{\texttt{reacq}}\frac{}{\Gamma \vdash \texttt{reacq}\ ad \in \texttt{Unit}}$$

Figure 3.11: Typing rules for concurrency constructs (inductive type `type_of` in `Sem.v`).

of a monitored reference using the $\langle\,\texttt{!}\,\rangle$ operator. A similar logic applies to assignment, because its typing rule ($\tau_{\texttt{asg}}$) applies only to writeable references.

Figure 3.11 lists the typing rules for concurrency related terms. An acquire term `acq` is well-typed if it is acquiring a monitored reference and if its body is well-typed within the SAFE version of its parent's environment. We also add two new variables in the typing environment of the body of the acquire: the variables `a` and `self`. The former comes from the function attached to the acquire, and the latter references the acquired monitored reference. Similar to when initializing monitored references, the usage of SAFE enforces that unsafe values do not leak into acquired monitored references. Additionally, the `acq` term has a safe type $\langle\,\sigma\,\rangle$, which prevents unsafe values from escaping from the monitored reference.

A critical region term `cr` is well-typed if its body is well-typed given the empty environment. As we discussed, we can use the empty environment for typing its subterm because `cr` (same as `init`) cannot appear inside the body of an acquire or of a function abstraction.

As with monitored reference initialization and `acq` terms, a `spawn` term is well-typed if its body is well-typed within a SAFE version of its parent's environment, so that unsafe values cannot not leak into new threads.

For `wait` terms, we check whether its subterm is a monitored reference. A `wait` only performs side-effects, so it types to `Unit`. A `reacq` does the same, so it also types to `Unit`.

3.4 Implementation

The operational semantics we defined is not an actual interpreter implementation for Elo. It is a formal description of how a interpreter should behave when evaluating Elo programs. We did implement a Elo interpreter in Coq, and proved that it matches the semantics of our formalizations. We implemented a

step function that evaluates threads individually, and then composed it with a round-robin scheduler, using monads to deal with blocked threads. You can check the Coq code for the interpreter in the `Eval.v` file.

The interpreter mainly serves as a way to test-drive the language, since it is (extremely) slow. It has no optimizations and does not run the threads in parallel, only concurrently. The implementation did force us to make some choices, though. For example, we had to chose which scheduling algorithm to use to manage the threads, and we implemented the round-robin scheduler for simplicity. This sheds some light to the fact that the operational semantics we described does not discriminate between scheduling algorithms. On the contrary, it allows us to implement any of them, as it should.

The notion that the semantics describes behavior and not implementation is vital if we are to understand some of the design choices we made when defining the step relations. For instance, earlier we mentioned that every memory cell has a lock, even those that have nothing to do with monitored references. The reason for this is simple: it simplifies our definitions without altering the behavior of the evaluation. Clearly, we could have defined two types of cells, with and without locks, and treated them accordingly. However, this would add a lot of cumbersome scaffolding to the step rules and the proofs, making them harder to read and understand. Any serious Elo implementation should optimize memory allocation by distinguishing between the two types of cells, or by storing locks some other way. Also, as mentioned, the reduction steps do not make use of the types and owners stored in the memory. An actual implementation should not store or even compute these pointer types and owners. Proving that this optimization does not alter the semantics of the language should be straightforward, since pointer types and owners do not appear as premises in any of the step rules: owners only ever get written to the memory, and pointer types only get written to the memory and passed around inside terms.

The evaluation of `await` terms also leaves ample space for optimizations. There is no constraint in the semantics stating that a waiting thread cannot immediately reacquire the lock after releasing it. This is undesirable, since the thread would simply go back to waiting after a few more steps. Naturally, in a real-world implementation, we only want to allow a waiting thread to reacquire a lock after a different thread releases it, which enables other threads to possibly change the state of the monitored reference that the original thread is waiting on.

Lastly, regarding the lock operations specifically, note that they must be atomic. This is not made explicit by the semantics, since each step is atomic

by itself. A real interpreter would have to use an atomic *unlock* operation for $\text{M}_{\texttt{rel}}$ and $\text{M}_{\texttt{wrel}}$, and another atomic operation for *locking if unlocked* for $\text{M}_{\texttt{acq}}$ and $\text{M}_{\texttt{wacq}}$. The latter could use an atomic instruction such as CAS (*compare-and-swap*).

4
Properties

In this chapter, we present some of the properties we use throughout our proofs. Even though we try to focus on the more abstract parts of the formalizations, we do discuss some of its technical aspects when we deem it relevant.

Some of the properties we define are quite simple. For instance, in our proofs, we sometimes need to know whether or not a thread has access to a memory address. In those cases, we use a property that checks for the presence of reference literals within the term that reifies the thread. We also use these basic properties to define more complex properties. For example, the property that guarantees that threads access monitored references in mutual exclusion uses two basic properties to assess when a thread is inside a critical region. Moreover, many of the properties we define are also invariant properties. An invariant property is valid throughout all the evaluation steps of a program. The mutual exclusion property we mentioned is an example of an invariant property.

It is worth pointing out that we define all properties using syntactical analysis or build them on top of other properties that perform these analyses. We are able to employ such a simple method because of a key characteristic of our formalizations: we store information about the state of the program inside the program. Specifically, we *enrich* the terms and the memory to hold information about the program. The following list describes these enrichments.

- The type τ in a reference literal $\langle\, ad : \tau \,\rangle$ is an enrichment. We added it so we could easily identify read-only, monitored, and writeable references.
- We could have implemented allocation and initialization using only the `new` term, which makes the very existence of the `init` term an enrichment. We added it because it allows us to check when a monitored reference is being initialized.
- The address ad in the term $\langle\, \texttt{cr}\ ad\ t \,\rangle$ is also an enrichment, since we could have implemented its functionality using simpler terms. For example, we could remove the address from the term, leaving it as $\langle\, \texttt{cr}\ t \,\rangle$, and then create a new term $\langle\, \texttt{release}\ ad \,\rangle$ that simply releases the lock of the

address *ad*. However, we use the address in the `cr` term to check when a thread is inside a critical region.

- Lastly, having a pointer type and an owner associated to every memory address is also an enrichment. As we will see, this allows us to define some properties more concisely.

These enrichments are inconsequential to the operational semantics of the language. Even those that alter how we formalize some concepts (`init` and `cr`) do so without changing the semantics behind the formalizations. It is precisely this that differentiates an enrichment from cheating to make the proofs easy. To illustrate, let us analyze term step rule for WRITE. It does not use the type τ from $\langle\, \texttt{(}\, ad : \tau \,\texttt{)} := v \,\rangle$; in fact, it ignores it completely. If the semantics were to use that type to check for writeable reference types, this would constitute what we called cheating. In a compiler implementation, this would be analogous to adding runtime checks to the language. Obviously, we do not want that, nor do we need it. Thus, we can consider enrichments as optimizations for the proofs.

4.1 Basic properties

Basic properties ensure the presence and/or the absence of a given subterm within a term, so we use them to analyze terms syntactically. As we will see, we also use some of the basic properties to compose more complex properties.

Property (`reserved-words` t) (file `Keywords.v`): the term t has no subterms that introduce `self` variables.

Property (`no-ref` ad t) (file `NoRef.v`): the term t has no $\langle\, ad : \tau \,\rangle$ subterms.

Property (`no-wref` ad t) (file `NoWRef.v`): the term t has no $\langle\, ad : \texttt{w\&}\tau \,\rangle$ subterms.

Property (`no-init` ad t) (file `NoInit.v`): the term t has no $\langle\, \texttt{init}\ \tau\ ad\ t' \,\rangle$ subterms.

Property (`no-cr` ad t) (file `NoCR.v`): the term t has no $\langle\, \texttt{cr}\ ad\ t' \,\rangle$ subterms.

Property (`no-reacq` ad t) (file `NoReacq.v`): the term t has no $\langle\, \texttt{reacq}\ ad \,\rangle$ subterms.

The last five properties are specific to a single memory address. The following properties generalize some of those definitions for all addresses.

$$\texttt{no-refs}\ t = \forall\, ad\,,\ \texttt{no-ref}\ ad\ t$$
$$\texttt{no-inits}\ t = \forall\, ad\,,\ \texttt{no-init}\ ad\ t$$
$$\texttt{no-crs}\ t = \forall\, ad\,,\ \texttt{no-cr}\ ad\ t$$
$$\texttt{no-reacqs}\ t = \forall\, ad\,,\ \texttt{no-reacq}\ ad\ t$$

Besides the "no" properties, that check for the *absence* of subterms within a term, we also define properties that check for the *presence* of subterms within a term.

Property (`one-init` ad t) (file `OneInit.v`): the term t has exactly one subterm $\langle\, \texttt{init}\ \tau\ ad\ t' \,\rangle$.

Property (`one-cr` ad t) (file `OneCR.v`): the term t has exactly one subterm $\langle\, \texttt{cr}\ ad\ t' \,\rangle$.

Definition. The *point* term is the next subterm of a term that is going to be reduced by a T–Step computation rule. For example, the point term of the arithmetic expression $\langle\, (1 + (2 - 3)) + 4 \,\rangle$ is the subterm $\langle\, 2 - 3 \,\rangle$.

Property (`waiting` ad t) (file `Waiting.v`): the term t has exactly one subterm $\langle\, \texttt{reacq}\ ad \,\rangle$, which is also the point term.

(We only use the notion of point terms to define the `waiting` property, but note that an `init` or `cr` subterm must always contain the point term of a thread. The `one-init` and `one-cr` properties could enforce this, but they do not. For practical reasons, we chose to express this idea through other properties and lemmas.)

The `one-cr` property ensures that a term is syntactically inside the critical region of a monitored reference. However, that is not enough to establish whether the term is holding the lock of that monitored reference, since the term could be waiting to reacquire the lock. For this reason, we created the `holding` and `not-holding` properties (`Holding.v`).

$$\texttt{holding}\ t\ = \texttt{one-cr}\ ad\ t\ \wedge\ \texttt{no-reacq}\ ad\ t$$
$$\texttt{not-holding}\ t\ = \texttt{no-cr}\ ad\ t\ \vee\ (\,\texttt{one-cr}\ ad\ t\ \wedge\ \texttt{waiting}\ ad\ t\,)$$

Note that these are memory-related properties that do not mention the memory (remember that locks are memory abstractions in Elo). This is one of the benefits of term enrichment. The `cr` and `reacq` terms contain the addresses

$$\frac{}{\texttt{no-cr}\ ad\ \texttt{unit}} \qquad \frac{}{\texttt{no-cr}\ ad\ n} \qquad \frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2}{\texttt{no-cr}\ ad\ (t_1\ \texttt{+}\ t_2)} \qquad \frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2}{\texttt{no-cr}\ ad\ (t_1\ \texttt{-}\ t_2)}$$

$$\frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2}{\texttt{no-cr}\ ad\ (t_1\texttt{;}\ t_2)} \qquad \frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2 \quad \texttt{no-cr}\ ad\ t_3}{\texttt{no-cr}\ ad\ (\texttt{if}\ t_1\ t_2\ t_3)} \qquad \frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2}{\texttt{no-cr}\ ad\ (\texttt{while}\ t_1\ t_2)}$$

$$\frac{}{\texttt{no-cr}\ ad\ x} \qquad \frac{\texttt{no-cr}\ ad\ t}{\texttt{no-cr}\ ad\ (\lambda\, a : \tau .\ t)} \qquad \frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2}{\texttt{no-cr}\ ad\ (t_1\ t_2)}$$

$$\frac{}{\texttt{no-cr}\ ad\ (ad' : \tau)} \qquad \frac{\texttt{no-cr}\ ad\ t}{\texttt{no-cr}\ ad\ (\texttt{new}\ \tau\ t)} \qquad \frac{\texttt{no-cr}\ ad\ t}{\texttt{no-cr}\ ad\ (\texttt{init}\ \tau\ ad'\ t)}$$

$$\frac{\texttt{no-cr}\ ad\ t}{\texttt{no-cr}\ ad\ (\texttt{!}\ t)} \qquad \frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2}{\texttt{no-cr}\ ad\ (t_1\ \texttt{:=}\ t_2)} \qquad \frac{\texttt{no-cr}\ ad\ t}{\texttt{no-cr}\ ad\ (\texttt{spawn}\ t)}$$

$$\frac{\texttt{no-cr}\ ad\ t_1 \quad \texttt{no-cr}\ ad\ t_2}{\texttt{no-cr}\ ad\ (\texttt{acq}\ t_1\ (\lambda x\ .\ t_2))} \qquad \frac{ad \neq ad' \quad \texttt{no-cr}\ ad\ t}{\texttt{no-cr}\ ad\ (\texttt{cr}\ ad'\ t)}$$

$$\frac{\texttt{no-cr}\ ad\ t}{\texttt{no-cr}\ ad\ (\texttt{wait}\ t)} \qquad \frac{}{\texttt{no-cr}\ ad\ (\texttt{reacq}\ ad')}$$

Figure 4.1: The constructors of the `no-cr` property (inductive type `no-cr` in `NoCR.v`).

associated with their monitored references; thus, we are able to check whether or not a thread is holding a lock by performing simple syntactical analyses.

4.1.1 Formalizations

In Coq, we defined the `reserved-words`, `no-ref`, `no-wref`, `no-init`, `no-cr`, `no-reacq`, `one-init`, `one-cr`, and `waiting` properties as inductive types (from the Calculus of Inductive Constructions [19]). In this text, we present inductive definitions as sets of structured inference rules. (For this reason, we use the terms "inference rule" and "constructor" interchangeably when referring to these concepts.)

Figure 4.1 lists all of the constructors for the `no-cr` property. As we can see, most of them repeat the same pattern: a given term satisfies the property if all of its subterms satisfy the property. We say that these are *structural* constructors. The single meaningful inference rule for `no-cr` is (unsurprisingly) the one regarding `cr` terms. It establishes that $\langle\, \texttt{no-cr}\ ad\ (\texttt{cr}\ ad'\ t) \,\rangle$ is true if and only if we have $\langle\, \texttt{no-cr}\ ad\ t \,\rangle$ and $\langle\, ad \neq ad' \,\rangle$. The first condition is structural, and the second condition guarantees that the term can be inside

$$\frac{\texttt{one-cr}\ ad\ t_1 \qquad \texttt{no-cr}\ ad\ t_2}{\texttt{one-cr}\ ad\ (t_1 + t_2)} \qquad \frac{\texttt{no-cr}\ ad\ t_1 \qquad \texttt{one-cr}\ ad\ t_2}{\texttt{one-cr}\ ad\ (t_1 + t_2)}$$

$$\frac{\texttt{no-cr}\ ad\ t}{\texttt{one-cr}\ ad\ (\texttt{cr}\ ad\ t)} \qquad \frac{ad \neq ad' \qquad \texttt{one-cr}\ ad\ t}{\texttt{one-cr}\ ad\ (\texttt{cr}\ ad'\ t)}$$

Figure 4.2: Some of the constructors of the `one-cr` property.

a critical region, but not inside the critical region monitored by the address *ad*. Note that there is no rule for when the addresses are equal. This is the case that does not satisfy the predicate, and we formalize it by not giving it a constructor. Thus, there is no way to "construct" the property for this undesired scenario.

We presented the complete formal definition for `no-cr` for the sake of thoroughness. The formalizations for `reserved-words`, `no-ref`, `no-wref`, `no-init`, and `no-reacq` repeat the same overhaul structure. Henceforth, we will only show and discuss the most important parts of the other formalizations, but mostly refrain from dwelling on them for space-saving reasons.

The structural constructors for the "no" properties ensure that *all* the subterms of a term satisfy a given condition, which mirrors the semantics of logic's *universal* quantifier $\langle\forall\rangle$. There is another kind of property, which ensures that *at least one* of the subterms of a term satisfies a given condition, mirroring the semantics of the *existential* quantifier $\langle\exists\rangle$. This is the idea behind the `one-init`, `one-cr`, and `waiting` definitions.

The formalizations for existential properties differ from what we have seen for universal properties like `no-cr`. Figure 4.2 showcases this difference in regard to some of the constructors for `one-cr` (a similar logic applies to `one-init` and `waiting`). In contrast to the `no-cr` property, that only has one constructor for addition, the `one-cr` property has two, and both of them guarantee that exactly one subterm is inside the critical region. We need to have two distinct constructors because either the first or the second subterm can be the one that is inside the critical region. Naturally, the omitted structural constructors follow the same reasoning. As to the non-structural constructors for `one-cr` (the ones that relate to the `cr` term), they also check for two cases: either the term is inside the desired critical region, or the term is inside a critical region that contains the desired critical region.

4.1.2 Preservation & Inheritance

Definition. Given a step $\langle t_1 \xrightarrow{e} t_2 \rangle$, property *preservation* states that if a property holds for t_1 then it also holds for t_2, while property *inheritance* states that if a property holds for t_2 then it also holds for t_1.

We proved preservation and inheritance lemmas for most of the previously defined basic properties. Depending on the property, however, it is natural that preservation/inheritance does not apply to some types of steps. For instance, if a term is not inside the critical region of a monitored reference and it steps to acquire that monitored reference, then the `no-cr` property is not preserved.

$$\begin{array}{ll} \texttt{no-cr}\ ad\ t_1 & \rightarrow \\ t_1 \xrightarrow{\text{ACQ}\ ad\ t} t_2 & \rightarrow \\ \neg\ \texttt{no-cr}\ ad\ t_2 & \end{array}$$

In fact, we can prove that the term enters the critical region.

$$\begin{array}{ll} \texttt{no-cr}\ ad\ t_1 & \rightarrow \\ t_1 \xrightarrow{\text{ACQ}\ ad\ t} t_2 & \rightarrow \\ \texttt{one-cr}\ ad\ t_2 & \end{array}$$

For these properties that are not invariably true, we still prove the preservation and inheritance lemmas we can given the proper conditions. In the case of the previous example, we can prove that preservation holds if we have different addresses for the critical region and the effect.

$$\begin{array}{ll} \texttt{no-cr}\ ad\ t_1 & \rightarrow \\ t_1 \xrightarrow{\text{ACQ}\ ad'\ t} t_2 & \rightarrow \\ ad \neq ad' & \rightarrow \\ \texttt{no-cr}\ ad\ t_2 & \end{array}$$

We proved these preservation and inheritance lemmas by case analysis. Specifically, by induction on the step operations, so that the induction hypothesis handles the cases that derive from evaluation contexts.

4.2 Invariant properties

An *invariant* property is a property that is preserved by any step that a program can take. We defined many invariant properties for Elo. (For convenience, we aggregated all of them in a single `invariants` predicate in the `Invariants.v` file.) The `reserved-words` property we covered earlier, for example, is also an invariant property. Proving its preservation theorems and the preservation theorems for the other invariants was a big part of the work for this thesis. We will not discuss these proofs, since they are long and also fairly straightforward. However, the conceptualization of what properties we needed to create to be able to prove what we wanted (and which of those were invariants) is much more difficult and important. For that reasoning, we will now discuss some of the core invariants of the language.

4.2.1 The `consistent-waits` property

We use the `consistent-waits` property (`ConsistentWaits.v`) to ensure that `wait` and `reacq` subterms can only be present and/or absent within certain parts of a term: in a term $\langle$ `acq` t_1 `(`λa `.` t_2`)` $\rangle$, the subterm t_2 can contain $\langle$ `wait self` $\rangle$ terms but no `reacq` terms; in a term $\langle$ `cr` ad $t\rangle$, the subterm t can contain $\langle$ `wait` $(ad : \tau)\rangle$ and $\langle$ `reacq` $ad\rangle$ terms; in a $\langle \lambda a : \tau . t\rangle$ or $\langle$ `spawn` $t\rangle$ term, the subterm t cannot contain `wait` and `reacq` terms. Naturally, a thread in the the thread pool can only contain `wait` and `reacq` terms that fit these rules.

4.2.2 The `consistent-regions` property

As we have seen, newly allocated writeable references have their owners stored in the memory by the reduction relations. The `consistent-regions` property (`ConsistentRegions.v`) guarantees that a writeable reference within a term is always in a region that belongs to the owner of that reference. This property is specially important to our final safety proof. It allows us to prove that the point term in a thread that reads from or writes to an address is in a region that belongs to the owner of that address.

Formally, a writable reference within a term must be inside of either a *thread* region or a *monitored* region. (A term can also be inside an *invalid* reference region, but these are a technicality of the reductions, so we will refrain from discussing them.)

r	:=		Reference Regions
	|	$\mathrm{R}_{\texttt{thread}}\ tid$	*thread region*
	|	$\mathrm{R}_{\texttt{monitored}}\ ad$	*monitored region*
	|	$\mathrm{R}_{\texttt{invalid}}$	*invalid region*

We delimit the boundaries between reference regions as we recursively analyze a term and its subterms. A term that represents a thread is always in a *thread* region. If a term contains a $\langle$ `init x&`τ ad $t\rangle$ or $\langle$ `cr` ad $t\rangle$ subterm, then t is within a *monitored* region for the memory address ad.

Because reference literals in the memory can themselves refer to other addresses, memory values must also have `consistent-regions`. We ascertain different things about a memory value depending on its pointer type: for a read-only reference type, the value must not contain writeable references (`no-wref`); for a monitored reference type, the value must be in a *monitored* region of the corresponding address; for a writeable reference type, the value must be in a region that matches that of the owner of the cell. This "memory invariant" is necessary for us to prove the `consistent-regions` invariant for threads.

4.2.3 The `unique-inits` property

Some properties apply to a memory and a set of threads. We use the auxiliary predicates *for all threads* ($\forall_{\texttt{threads}}$) and *for one thread* ($\exists!_{\texttt{thread}}$) to define them. The $\langle \forall_{\texttt{threads}} \rangle$ predicate states that all the threads in a given thread pool satisfy a given property. The $\langle \exists!_{\texttt{thread}} \rangle$ predicate states that, given a property $\mathtt{P}_1$ and a property $\mathtt{P}_2$, exactly one thread in a given thread pool satisfies $\mathtt{P}_1$ while all the other threads satisfy $\mathtt{P}_2$.

Property (`unique-inits` m ths) (file `UniqueInits.v`)**.** For all allocated cells in the memory, if a cell is not empty, then no threads are in the process of initializing it; symmetrically, if a cell is empty, then only one thread is in the process of initializing it.

$$\begin{aligned}&\texttt{unique-inits}\ m\ ths = \forall\, ad\,,\, ad < \#m\ \rightarrow\\ &\quad (\, m[ad].t \neq \texttt{none}\ \rightarrow\ \forall_{\texttt{threads}}\ ths\ (\texttt{no-init}\ ad)\,)\ \wedge\\ &\quad (\, m[ad].t = \texttt{none}\ \rightarrow\ \exists!_{\texttt{thread}}\ ths\ (\texttt{one-init}\ ad)\ (\texttt{no-init}\ ad)\,)\end{aligned}$$

Remember that `no-init` and `one-init` are properties that range over an address and a term. The $\langle \forall_{\texttt{threads}} \rangle$ and $\langle \exists!_{\texttt{thread}} \rangle$ quantifiers expect properties over terms, so we curry the definitions by providing only the address, as in $\langle$ `no-init` $ad\rangle$ and $\langle$ `one-init` $ad\rangle$.

We mainly use the `unique-inits` property to prove that we cannot initialize the same memory address twice, and to prove that a memory cell never goes back to being empty once we place a term inside it. We use these lemmas in our final safety proof.

4.2.4 The `mutual-exclusion` property

Property (`mutual-exclusion` m ths) (file `MutualExclusion.v`)**.** For all memory addresses, if an address is unlocked, then no threads are holding its lock; symmetrically, if an address is locked, then exactly one thread is holding its lock.

$$\texttt{mutual-exclusion}\ m\ ths = \forall\, ad\,,$$
$$(\, m[ad].\mathcal{X} = \texttt{false} \rightarrow \forall_{\texttt{threads}}\ ths\ (\texttt{not-holding}\ ad\,)\,) \wedge$$
$$(\, m[ad].\mathcal{X} = \texttt{true} \rightarrow \exists!_{\texttt{thread}}\ ths\ (\texttt{holding}\ ad\,)\ (\texttt{not-holding}\ ad\,)\,)$$

The `mutual-exclusion` property is one of the cornerstones of our final safety proof. Despite its complex appearance, its preservation theorems mostly derive from the semantics of the language.

4.3 Initial properties

Definition. An *initial* property is a property that programs must satisfy before the beginning of the evaluation.

A program in Elo as provided by the programmer must satisfy some basic initial properties: it must not contain intermediary terms (references and `init`/`cr`/`reacq` terms); it must not introduce reserved words; it must have consistent `wait` terms; and it must be well typed (a term is well typed if it has a type given the empty environment). These are very basic properties that can be trivially checked by any compiler. That is to say that Elo's initial requirements are in par with standard requirements of other programming languages.

For convenience, we aggregated the initial properties in a single predicate.

$$\begin{aligned}\texttt{initial}\ t = {} & \texttt{no-refs}\ t \\ & \wedge \texttt{no-inits}\ t \\ & \wedge \texttt{no-crs}\ t \\ & \wedge \texttt{no-reacqs}\ t \\ & \wedge \texttt{reserved-words}\ t \\ & \wedge \texttt{consistent-waits}\ t \\ & \wedge \texttt{well-typed-term}\ t\end{aligned}$$

As we will see, the `initial` property appears as the single hypothesis in our safety theorem. Since we proved our theorems in Coq, if you want to ascertain whether or not our final proof is correct, than the initial properties, the language semantics, and the definition for data races (which we will provide in the next chapter) are all that you need to check. Every other property we defined is a part of the inner workings of our proofs. We only presented and discussed them so you can better understand the general ideas behind the proofs.

5
Safety

In this chapter, we prove that a valid and well-typed Elo program can never produce a data race. We define data races in relation to events and traces. We say that a trace contains a data race when some of its events overlap in time. Like other data-race definitions [16, 17], we use the *happens-before* relation to express that two events cannot overlap in time. The following definitions formalize these concepts in regard to pairs of events in a trace.

Definition. Two events *synchronize* with each other if the first event initializes or releases an address and the second event acquires that address.

Definition. An event *happens before* another event if both execute on the same thread or if they synchronize with each other. Moreover, if an event x happens before an event y and the event y happens before an event z, then the event x happens before the event z. That is, *happens-before* is a transitive relation.

Definition. Two events *conflict* with each other if one of them writes to a memory address and the other reads or writes to that same memory address.

Definition. Two conflicting events form a *data race* unless one of them happens before the other.

We formalize the *happens-before* and *data-race* definitions in Coq by creating two inductive properties that range over traces.

Figure 5.1 lists the four constructors for the `happens-before` property. To facilitate our proofs, we define the property in regard to the events located on the edges of a trace. Remember that traces are reversed, so the notation $\langle\, [\, ev_2 \,] +\!\!+ tc +\!\!+ [\, ev_1 \,] \,\rangle$ represents a trace that starts with the event ev_1, follows up with zero or more events in the inner trace tc, and ends with the event ev_2. The first `happens-before` constructor (`HBthread`) handles events from the same thread; the second constructor (`HBinit`) handles the synchronization case between initializing and acquiring a monitored reference; the third constructor (`HBlock`) handles the synchronization case between releasing and acquiring a monitored reference; the fourth constructor (`HBtrans`) handles transitivity. Moreover, the second constructor uses the `is-initialize` predicate to check for the INIT effect and the `is-acquire` predicate to check for the acquire effects (ACQ and WACQ). The third constructor also uses `is-acquire`, and it uses the `is-release` predicate to check for the release effects (REL and WREL).

$$\text{HB}_{\texttt{thread}}\ \frac{}{\texttt{happens-before}\ (\ [\,(tid,\ e_2)\,] +\!\!+ tc +\!\!+ [\,(tid,\ e_1)\,]\ )}$$

$$\text{HB}_{\texttt{init}}\ \frac{\texttt{is-initialize}\ ad\ e_1 \quad \texttt{is-acquire}\ ad\ e_2}{\texttt{happens-before}\ (\ [\,(tid_2,\ e_2)\,] +\!\!+ tc +\!\!+ [\,(tid_1,\ e_1)\,]\ )}$$

$$\text{HB}_{\texttt{lock}}\ \frac{\texttt{is-release}\ ad\ e_1 \quad \texttt{is-acquire}\ ad\ e_2}{\texttt{happens-before}\ (\ [\,(tid_2,\ e_2)\,] +\!\!+ tc +\!\!+ [\,(tid_1,\ e_1)\,]\ )}$$

$$\text{HB}_{\texttt{trans}}\ \frac{\begin{array}{c}\texttt{happens-before}\ (\ [\,ev_2\,] +\!\!+ tc_a +\!\!+ [\,ev_1\,]\ ) \\ \texttt{happens-before}\ (\ [\,ev_3\,] +\!\!+ tc_b +\!\!+ [\,ev_2\,]\ )\end{array}}{\texttt{happens-before}\ (\ [\,ev_3\,] +\!\!+ tc_b +\!\!+ [\,ev_2\,] +\!\!+ tc_a +\!\!+ [\,ev_1\,]\ )}$$

Figure 5.1: The `happens-before` property (in file `HappensBefore.v`).

$$\text{DR}_{\texttt{rw}}\ \frac{\begin{array}{c}ev_1 = (tid_1,\ \textsc{read}\ \ ad\ \ t_1) \\ ev_2 = (tid_2,\ \textsc{write}\ \ ad\ \ t_2) \\ \neg\ \texttt{happens-before}\ (\ [\,ev_2\,] +\!\!+ tc +\!\!+ [\,ev_1\,]\ )\end{array}}{\texttt{data-race}\ (\ [\,ev_2\,] +\!\!+ tc +\!\!+ [\,ev_1\,]\ )}$$

$$\text{DR}_{\texttt{wr}}\ \frac{\begin{array}{c}ev_1 = (tid_1,\ \textsc{write}\ \ ad\ \ t_1) \\ ev_2 = (tid_2,\ \textsc{read}\ \ ad\ \ t_2) \\ \neg\ \texttt{happens-before}\ (\ [\,ev_2\,] +\!\!+ tc +\!\!+ [\,ev_1\,]\ )\end{array}}{\texttt{data-race}\ (\ [\,ev_2\,] +\!\!+ tc +\!\!+ [\,ev_1\,]\ )}$$

$$\text{DR}_{\texttt{ww}}\ \frac{\begin{array}{c}ev_1 = (tid_1,\ \textsc{write}\ \ ad\ \ t_1) \\ ev_2 = (tid_2,\ \textsc{write}\ \ ad\ \ t_2) \\ \neg\ \texttt{happens-before}\ (\ [\,ev_2\,] +\!\!+ tc +\!\!+ [\,ev_1\,]\ )\end{array}}{\texttt{data-race}\ (\ [\,ev_2\,] +\!\!+ tc +\!\!+ [\,ev_1\,]\ )}$$

Figure 5.2: The `data-race` property (in file `Safety.v`).

Figure 5.2 formalizes the `data-race` property. As with `happens-before`, we characterize a data race in regard to a trace and the events located on its edges. There are three kinds of conflicting events in a trace: READ–WRITE, WRITE–READ, and WRITE–WRITE, all with respect to the same address. Each `data-race` constructor addresses one of these cases and ensures the absence of a happens-before ordering.

With that, we state our *safety* theorem as follows.

$$\begin{array}{ll}(\text{H}_{\texttt{init}}) & \texttt{initial}\ t \quad \rightarrow \\ (\text{H}_{\texttt{ustep}}) & [\,]\ /\ [\,t\,] \ \overset{tc_3 +\!\!+ tc_2 +\!\!+ tc_1}{\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow\!\!\triangleright^*}\ m\ /\ ths \ \rightarrow \\ & \neg\ (\ \texttt{data-race}\ tc_2\ )\end{array}$$

The theorem contemplates the evaluation of a correct program, that is, one that satisfies Elo's initial properties. The term t stands for that program as provided by the programmer, while `[]` represents the empty memory at

the beginning of execution, and `[`t`]` represents the initial thread pool with its single thread. In order to express that the trace of an evaluation does not contain a data race, we assert that no matter how we break the multistep trace (tc_1, tc_2, and tc_3), the subtrace tc_2 does not constitute a data race. Since the break points are arbitrary, tc_2 can be any subtrace within the whole trace.

Note that the subtrace tc_3 is irrelevant to our proof, since a multistep relation does not necessarily represent the evaluation of a program to completion. On the contrary, a multistep relation holds for all intermediary steps in the evaluation, and it does not enforce that the memory m and the thread pool ths in the theorem represent the final state of the program. Thus, we can state the *safety* theorem by using only tc_1 and tc_2, and our proof will be just as powerful.

$$
\begin{array}{ll}
(\textsc{H}_{\texttt{init}}) & \texttt{initial}\ t \quad \rightarrow \\
(\textsc{H}_{\texttt{ustep}}) & \texttt{[]}\ /\ \texttt{[}t\texttt{]} \overset{tc_2 +\!\!+ tc_1}{\rightsquigarrow\!\!\triangleright^{*}}\ m\ /\ ths \quad \rightarrow \\
 & \neg\ (\ \texttt{data-race}\ tc_2\ )
\end{array}
$$

Let us now begin to prove the theorem. By breaking the multistep reduction from $\textsc{H}_{\texttt{ustep}}$ in two parts, we get an extra discrete point in the timeline of the reduction, besides the initial and the final ones. We use the suffix $\langle\, x \,\rangle$ to identify the state of the program after the tc_1 multistep, and the suffix $\langle\, y \,\rangle$ to identify the state of the program after the tc_2 multistep. (This effectively renames m and ths to m_y and ths_y). We also use $\langle\, 0 \,\rangle$ to label the beginning of the reduction.

$$
\begin{array}{ll}
(\textsc{H}_{\texttt{init}}) & \texttt{initial}\ t \quad \rightarrow \\
(\textsc{H}_{\texttt{ustep0X}}) & \texttt{[]}\ /\ \texttt{[}t\texttt{]} \overset{tc_1}{\rightsquigarrow\!\!\triangleright^{*}}\ m_x\ /\ ths_x \quad \rightarrow \\
(\textsc{H}_{\texttt{ustepXY}}) & m_x\ /\ ths_x \overset{tc_2}{\rightsquigarrow\!\!\triangleright^{*}}\ m_y\ /\ ths_y \quad \rightarrow \\
 & \neg\ (\ \texttt{data-race}\ tc_2\ )
\end{array}
$$

From $\textsc{H}_{\texttt{initial}}$ and then from the preservation of the invariants, we get that the `invariant` property holds in all points of the reduction. These new hypotheses replace $\textsc{H}_{\texttt{init}}$ and $\textsc{H}_{\texttt{ustep0X}}$, so we remove them. Besides that, we unfold the negation $\langle \neg \rangle$ from the conclusion so we can introduce the $\textsc{H}_{\texttt{race}}$

hypothesis.

$$
\begin{array}{ll}
(\text{H}_{\texttt{invX}}) & \texttt{invariants}\ m_x\ ths_x \rightarrow \\
(\text{H}_{\texttt{invY}}) & \texttt{invariants}\ m_y\ ths_y \rightarrow \\
(\text{H}_{\texttt{ustepXY}}) & m_x \mathbin{/} ths_x \overset{tc_2}{\rightsquigarrow}{}^{*}\ m_y \mathbin{/} ths_y \rightarrow \\
(\text{H}_{\texttt{race}}) & \texttt{data-race}\ tc_2 \rightarrow \\
 & \texttt{False}
\end{array}
$$

We then split the proof in three cases, one for each of the `data-race` constructors of $\text{H}_{\texttt{race}}$. In this text, we will only discuss the READ–WRITE case, since the proofs for the other two follow a similar logic.

$$
\begin{array}{ll}
(\text{H}_{\texttt{invX}}) & \texttt{invariants}\ m_x\ ths_x \rightarrow \\
(\text{H}_{\texttt{invY}}) & \texttt{invariants}\ m_y\ ths_y \rightarrow \\
(\text{H}_{\texttt{ustepXY}}) & m_x \mathbin{/} ths_x \overset{[ev_2] +\!\!+ tc +\!\!+ [ev_1]}{\rightsquigarrow}{}^{*}\ m_y \mathbin{/} ths_y \rightarrow \\
(\text{H}_{\texttt{ev1}}) & ev_1 = (tid_1,\ \textsc{read}\ \ ad\ \ t_1) \rightarrow \\
(\text{H}_{\texttt{ev2}}) & ev_2 = (tid_2,\ \textsc{write}\ \ ad\ \ t_2) \rightarrow \\
 & \texttt{happens-before}\ (\ [ev_2] +\!\!+ tc +\!\!+ [ev_1]\ )
\end{array}
$$

The rewritten proof goal reflects that the READ event must happen before the WRITE event. Continuing with the proof, we break the the $\text{H}_{\texttt{ustepXY}}$ hypotheses in three so we can analyze the reductions in regard to the edges of the trace. This introduces two other discrete points in the timeline of the reduction: $\langle\, a \,\rangle$ labels the program just after the the READ and $\langle\, b \,\rangle$ labels the program just before the WRITE. Naturally, the invariants also hold for these

states.

$$
\begin{array}{lll}
(\mathrm{H}_{\texttt{invX}}) & \texttt{invariants}\ m_x\ ths_x & \rightarrow \\
(\mathrm{H}_{\texttt{invA}}) & \texttt{invariants}\ m_a\ ths_a & \rightarrow \\
(\mathrm{H}_{\texttt{invB}}) & \texttt{invariants}\ m_b\ ths_b & \rightarrow \\
(\mathrm{H}_{\texttt{invY}}) & \texttt{invariants}\ m_y\ ths_y & \rightarrow \\
(\mathrm{H}_{\texttt{ev1}}) & ev_1 = (tid_1,\ \textsc{read}\ \ ad\ \ t_1) & \rightarrow \\
(\mathrm{H}_{\texttt{ev2}}) & ev_2 = (tid_2,\ \textsc{write}\ \ ad\ \ t_2) & \rightarrow \\
(\mathrm{H}_{\texttt{cstepXA}}) & m_x \mathbin{/} ths_x \overset{ev_1}{\rightsquigarrow\!\triangleright} m_a \mathbin{/} ths_a & \rightarrow \\
(\mathrm{H}_{\texttt{ustepAB}}) & m_a \mathbin{/} ths_a \overset{tc}{\rightsquigarrow\!\triangleright^*} m_b \mathbin{/} ths_b & \rightarrow \\
(\mathrm{H}_{\texttt{cstepBY}}) & m_b \mathbin{/} ths_b \overset{ev_2}{\rightsquigarrow\!\triangleright} m_y \mathbin{/} ths_y & \rightarrow \\
 & \texttt{happens-before}\ (\ [\,ev_2\,] +\!\!+ tc +\!\!+ [\,ev_1\,]\ ) &
\end{array}
$$

When tid_1 and tid_2 are equal, we trivially close the proof using the first constructor from the `happens-before` property. The other case leaves us with a $\langle\, tid_1 \neq tid_2 \,\rangle$ hypothesis. Moreover, the WRITE effect implies that the pointer type of the address ad is a writeable reference type. Because pointer types do not change, this is true for all memory states.

The general ideal of the rest of the proof goes as follows.

- Because the thread tid_1 is performing a READ on the address ad in point $\langle\, x \,\rangle$, we know that tid_1 is in a region owned by the address ad (as explained when discussed the `consistent-regions` property).
- Since READ steps do not alter the region of a thread, the thread tid_1 in point $\langle\, a \,\rangle$ is still in the same region that it was in point $\langle\, x \,\rangle$.
- In a similar manner, because the thread tid_2 is performing a WRITE on the address ad in point $\langle\, b \,\rangle$, we know that tid_2 is in a region owned by the address ad.
- Given that owners in the memory are static, we infer that the thread tid_1 in point $\langle\, a \,\rangle$ is in the same region that the thread tid_2 is in point $\langle\, b \,\rangle$. For the rest of the proof, we will only care for these two points in the reduction.
- By definition, the region the threads are in can be one of three: $\mathrm{R}_{\texttt{invalid}}$, $\mathrm{R}_{\texttt{thread}}$, or $\mathrm{R}_{\texttt{monitored}}$. This splits the proof in three cases. The *invalid* case is trivial because no valid memory contains an invalid owner. The *thread* case is also trivial, since different threads cannot be in the same thread region. The interesting case is the *monitored* case.

- A thread that is in a monitored region can either be initializing a monitored reference ad_x or inside the critical region of that monitored reference. (Note that the address ad_x is different from the address ad that we read and write from.) By applying this rationale to the threads tid_1 and tid_2 in $\langle\, a \,\rangle$ and $\langle\, b \,\rangle$, we split the proof in four cases:
  - If tid_1 in $\langle\, a \,\rangle$ is initializing ad_x and tid_2 in $\langle\, b \,\rangle$ is also initializing ad_x, we have a contradiction since two distinct threads cannot initialize the same reference (remember the `unique-inits` property).
  - If tid_1 in $\langle\, a \,\rangle$ is inside the critical region of ad_x and the thread tid_2 in $\langle\, b \,\rangle$ is initializing ad_x, we also have a contradiction. It is easy to prove that we cannot acquire a reference before initializing it.
  - The proof gets more complex if tid_1 in $\langle\, a \,\rangle$ is initializing ad_x or is inside its critical region, while tid_2 in $\langle\, b \,\rangle$ is inside the ad_x critical region.

Because the proof for both of the *initialize-then-acquire* and *release-then-acquire* cases follow a very similar structure, we will only discuss the *release-then-acquire* scenario.

- If tid_1 in $\langle\, a \,\rangle$ and tid_2 in $\langle\, b \,\rangle$ are both inside the ad_x critical region, we prove the happens-before goal as follows.
  - First, we assert that the threads are not only inside the critical region of the monitored reference (not only they contain a `cr` term), but they are also `holding` the lock of the monitored reference. Remember that a term is `holding` the lock of an address if it is inside its critical region and if it is not trying to reacquire the lock of the address. It is easy to prove that a thread that reads from or writes to an address cannot be trying to reacquire a lock.
  - Given that the thread tid_1 is holding the ad_x lock in point $\langle\, a \,\rangle$ and that the thread tid_2 is holding the ad_x lock in point $\langle\, b \,\rangle$, we know that the thread tid_1 must release the ad_x lock at some point in the reduction. (Formally we prove that by forward induction on the $\textsc{Hustep}_{\textsc{AB}}$ hypothesis). Otherwise, we would get a contradiction since two threads cannot hold the same lock simultaneously, as per the `mutual-exclusion` invariant. Let us label the point in the reduction that tid_1 releases the lock as $\langle\, p \,\rangle$.
  - In $\langle\, p \,\rangle$, we know that tid_1 does not have the lock of the monitored reference, and neither does any other thread, but we do know that somewhere along the evaluation from $\langle\, p \,\rangle$ to $\langle\, b \,\rangle$, the thread tid_2

acquires the lock. (We prove that by inducing backwards from $\langle b \rangle$ to $\langle p \rangle$.) Let us label the point in the reduction that tid_2 acquires the lock as $\langle v \rangle$. (Note that we are not interested in the many acquire-and-release cycles that can by performed on this monitored reference between $\langle p \rangle$ to $\langle v \rangle$. These operations are irrelevant to our goal of proving the `happens-before` property.)

– Finally, we have four sequential points in the timeline of the reduction that characterize the desired happens-before ordering: $\langle a \rangle$ is the point when tid_1 reads from ad, $\langle p \rangle$ is the point when tid_1 releases ad_x, $\langle v \rangle$ is the point when tid_2 acquires ad_x, and $\langle b \rangle$ is the point when tid_2 writes to ad. Naturally, $\langle p \rangle$ happens before $\langle v \rangle$ because of the lock synchronization rule (HBlock). We get that $\langle a \rangle$ happens before $\langle p \rangle$ because the associated events happen in the same thread tid_1. For the same reason, but with the thread tid_2, we get that $\langle v \rangle$ happens before $\langle b \rangle$. Thus, by transitivity, we conclude that the READ in $\langle a \rangle$ happens before the WRITE in $\langle b \rangle$.

Let us recapitulate on the safety theorem we just proved.

$$
\begin{array}{lll}
(\mathrm{H}_{\texttt{init}}) & \texttt{initial}\ t & \rightarrow \\
(\mathrm{H}_{\texttt{ustep}}) & \texttt{[]}\ /\ \texttt{[}t\texttt{]} \overset{tc_3 \mathbin{+\!\!+} tc_2 \mathbin{+\!\!+} tc_1}{\rightsquigarrow^*} m\ /\ ths & \rightarrow \\
 & \neg\ (\texttt{data-race}\ tc_2\,) &
\end{array}
$$

We have explained *how* we proved this theorem, but remember that given that we did it in Coq, the *how* is irrelevant to the validity of the proof. You only need to validate the concepts presented by the theorem in order to validate the proof. Let us (re)analyze these concepts: the `initial` predicate aggregates basic language properties, such as well-typedness, that can be easily checked by a compiler; the multistep that produces the traces is part of the formal semantics of the language, so you must believe the semantics is correct—and that it produces traces correctly—in order to believe in the proof; lastly, the definition of the `data-race` property comes directly from standard definitions used by other programming languages. The theorem concludes that no segment of the trace can contain a data race. Thus, as the concepts present in the theorem are fairly straightforward, it stands to reason that the language is safe from data races.

6
Related Work

As we mentioned before, monitors were first proposed by Brinch Hansen [9] and Hoare [10], with exchanges between them leading to the specification of the construct as we know it today. Despite looking for research involving the formalization of monitors, we could not find any relevant work pertaining to the topic. Therefore, in this chapter, we will discuss programming languages inspired by monitors and the formalization of concurrent programming languages in general.

Guava [30] is a dialect of Java that implements monitors with proper safety guarantees. Like Elo, it forbids threads from sharing unsynchronized mutable data, and therefore ensures the absence of data races. Guava has three categories of classes: monitors, objects, and values. They distinguish between one another in regard to concurrency as follows: *monitors* can be referenced from multiple threads, and enforce the expected mutual exclusion semantics; *objects* are restricted to be shared within a single thread at a time, and, thus, do not require synchronization; lastly, *values* are non-references that always get recursively copied when shared between threads, which eliminates any synchronization requirements. Guava deals with the "reference problem" we mentioned earlier by using a Rust-like ownership system that keeps track of owned and borrowed references within user-defined regions.

Guava was the only programming language we found that implemented monitors with references keeping its original data-race freedom guarantees. It also expanded on the abstraction by using interesting mechanisms to deal with the shortcomings of the original proposals. Unlike Elo, Guava is still restricted by the object-oriented modularity of Java, so some of its innovations cannot be applied to monitored references. Also unlike Elo, Guava does not have a formal specification and safety proof. The rules regarding ownership and regions, however, could be applied to Elo in order to loosen some of the restrictions regarding how mutable data is shared between threads.

As to the formalization of concurrent programming languages, in Volume 2 of Software Foundations [21], the textbook reference for operational semantics in Coq, the *Smallstep* chapter presents a $\langle\, t_1 \parallel t_2 \,\rangle$ term that reduces non-deterministically, meaning that a congruence step can reduce either t_1 or

t_2. Despite its simplicity, we found hard to prove actual properties using this formalization. Also, Volume 6, which introduces Separation Logic, does not cover reasoning about concurrent programs, even though Concurrent Separation Logic has been used to that purpose [27].

Rust [4] is, perhaps, the most prominent example of a programming language that uses Concurrent Separation Logic to prevent data races. As of today (May 2025), Rust still does not have an informal specification, much less a formal one. This becomes a problem when each of its research papers have to decide what Rust *is* before starting to prove anything [5, 6, 7, 8]. Moreover, one can argue that Rust's ownership system is complex in nature, and requires programmers to learn about a whole new set of features in order to be able to write basic code. Thus, the benefits of data-race-free code does not come without its hindrances.

The inspiration to structure Elo's reduction relations using effects came from Type and Effect Systems [22]. These systems structure typing judgments such that a type plus an effect associate with a term relative to a type environment. An effect typically describes program behavior during evaluation, as in "read from address x", "write to address y", and "raise exception z". Static analysis algorithms can then use this information to check for core language properties. Using Type and Effect Systems to deal with concurrency is not a novelty [23, 24]. However, as we have seen, Elo effects operate in a different abstraction level. We use effects to *prove* language properties instead of *enforcing* them, and we tie effects to reduction relations, not to typing judgments.

Despite our initial inspiration, Labeled Transition Systems (LTS) [20] relate to our formalizations much more closely than Type and Effect Systems. In LTS, transitions are defined as a subset of $S \times L \times S$, given that S is a set of states and L is a set of labels. We can then represent a transition from state x to state y with a label α as $\langle\, x \xrightarrow{\alpha} y \,\rangle$. CompCertTSO [25], a verified compiler for a concurrent C-like language, uses these principles when defining its small-step operational semantics. Their method for formalizing layered and modular semantics much resembles our approach with effects and the three reduction relations. Being a C-like language, however, CompCertTSO is not free from data races, so its formalization is not concerned with proving safety properties. As another example, we found a study [26] that uses LTS to provide a structural operational semantics for Python. Again, their focus lies not in proving language properties, but in extracting an interpreter from the formal semantics.

7
Conclusion

In this thesis, we have presented a way to formalize the monitor concept by decomposing it in terms of its core features: class-like modularity and data encapsulation through mutually exclusive accesses. We focused on the latter since that is the aspect of the model responsible for the safety guarantees that we were interested in proving. For our operational semantics, we defined three reduction relations with distinct semantic purposes and layered them cohesively to compose our single-step relation. We used effects to bridge between the concerns of the different layers and to allow for a true modular design.

The Coq code that implements our proofs and formalizations contains 13019 lines of code spread across 45 different files. Conceptually, we divided the proofs and definitions of the project into three categories. The *core* of the codebase mainly contains the semantics of Elo, the *properties* section contains Elo's many initial, invariant, and/or auxiliary properties, and the *safety* section deals with the end-goal proofs and definitions explained in Chapter 5.

As mentioned, we use lists (with some array-like operations) to represent the memory and the thread pool in Elo. Because of this, many of our proofs contain list boilerplate that is orthogonal to the ideas of what we are actually trying to prove. For example, it is common to see something like $\langle\, ths[tid \leftarrow t_1][tid] = t_2 \,\rangle$ as a proof hypothesis. It is obvious that we can rewrite that hypothesis as $\langle\, t_1 = t_2 \,\rangle$, but having to do it manually every single time for dozens of proofs is a hindrance. For that reason, we created some generic Coq tactics to deal with list manipulation. Specifically, the `sigma` and `omicron` tactics defined in the file `Array.v` use Coq's pattern matching capabilities in order to simplify obvious list-related hypotheses such as the one in the example.

A great deal of the development time spent to complete this project was spent proving property preservation. The most difficult part of the process was to formulate adequate properties for the concepts we wanted to express. With each new invariant we added, we had to prove the preservation (and sometimes inheritance) lemmas and theorems associated with it. As we explained, the process of proving property preservation consists of proving that a program that satisfies a property still satisfies it after a single execution step. For that,

we need to prove the invariance in regard to each of Elo's ten effects. These proofs were not particularly difficult, but they were time consuming. Some of them were repetitive, so we were able to use Coq tactics to cut down on the amount of boilerplate code in the proofs.

We judge that an important contribution of our work is the exposition of how we engineered the safety proof. Formal proofs can be hard to develop, even harder under the watchful eyes of a proof assistant, and to the best of our knowledge there is no established way to formalize and prove properties about concurrency for multithreaded languages. In that sense, ours was also an effort in proof engineering: we demonstrated how one can define commonplace concepts such as *immutability* and *data races*, and build readable and straightforward proofs using these definitions.

Another important contribution of this thesis relates to the use of enrichments, since we have not seen other formalizations use them as we did. By annotating the memory and the syntax with information about the state of the program, we were able to prove safety using mostly syntactical properties. This technique is specially valuable because it allowed us to structure most of our intermediary proofs without needing to rely on the evaluation trace. Our invariants only depend on the state of the system (memory and thread pool), which greatly simplifies their definitions.

For the future, we would like to expand our language with other core constructs, such as records, so we can properly emulate monitors. It would also be interesting to approximate our semantics to hardware level, by porting it to some kind of bytecode architecture, and then proving the necessary equivalences.

Bibliography

[1] SANTOS, R.. **Elo in Coq**. `https://github.com/renan061/elo-coq`, 2025.

[2] SANTOS, R.. **Revisiting Monitors**. Master's dissertation, PUC-Rio, 2020.

[3] SANTOS, R.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Revisiting monitors**. Science of Computer Programming, 196, 2020.

[4] KLABNIK, S.; NICHOLS, C.. **The Rust Programming Language, 2nd Edition**. No Starch Press, USA, 2022.

[5] REED, E. C.. **Patina : A formalization of the Rust programming language**, 2015.

[6] JUNG, R.; JOURDAN, J.-H.; KREBBERS, R. ; DREYER, D.. **RustBelt: Securing the foundations of the Rust programming language**. POPL, 2017.

[7] PEARCE, D. J.. **A lightweight formalism for reference lifetimes and borrowing in Rust**. ACM Trans. Program. Lang. Syst., 2021.

[8] WEISS, A.; GIERCZAK, O.; PATTERSON, D. ; AHMED, A.. **Oxide: The essence of Rust**, 2021.

[9] HANSEN, P. B.. **Operating System Principles**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[10] HOARE, C. A. R.. **Monitors: An operating system structuring concept**. Communications of the ACM, 17(10):549–557, October 1974.

[11] HANSEN, P. B.. **The programming language Concurrent Pascal**. IEEE Transactions on Software Engineering, SE-1, 1975.

[13] BUHR, P. A.; HARJI, A. S.. **Implicit-signal monitors**. ACM Trans. Program. Lang. Syst., 27(6):1270–1343, November 2005.

[14] ADVE, S.. **Data races are evil with no exceptions: technical perspective**. Commun. ACM, 2010.

[15] BOEHM, H.-J.; ADVE, S. V.. **You don't know jack about shared variables or memory models**. Commun. ACM, 2012.

[16] MANSON, J.; PUGH, W. ; ADVE, S. V.. **The Java memory model**. SIGPLAN Not., 40(1), 2005.

[17] C++ REFERENCE CONTRIBUTORS. **Memory model (threads and data races)**, 2019. [Online; accessed 21-January-2025].

[18] THE COQ DEVELOPMENT TEAM. **The Coq reference manual – release 8.19.0**. `https://coq.inria.fr/doc/V8.20.0/refman`, 2024.

[19] BERTOT, Y.; CASTERAN, P.. **Interactive Theorem Proving and Program Development**. Springer Berlin Heidelberg, 2004.

[20] PLOTKIN, G.. **A structural approach to operational semantics**. Journal of Logical and Algebraic Methods in Programming, 60-61, 2004.

[21] PIERCE, B.; OTHERS. **Software foundations**. `https://softwarefoundations.cis.upenn.edu`.

[22] NIELSON, F.; NIELSON, H. R.. **Type and effect systems**. In: CORRECT SYSTEM DESIGN, 1999.

[23] AMTOFT, T.; NIELSON, H. R. ; NIELSON, F.. **Type and Effect Systems – Behaviours for Concurrency**. Imperial College Press, 1999.

[24] BOCCHINO, R. L.; ADVE, V. S.; DIG, D.; ADVE, S. V.; HEUMANN, S.; KOMURAVELLI, R.; OVERBEY, J.; SIMMONS, P.; SUNG, H. ; VAKILIAN, M.. **A type and effect system for deterministic parallel Java**. SIGPLAN Not., 44(10), 2009.

[25] ŠEVČÍK, J.; VAFEIADIS, V.; ZAPPA NARDELLI, F.; JAGANNATHAN, S. ; SEWELL, P.. **CompCertTSO: A verified compiler for relaxed-memory concurrency**. J. ACM, 60(3), 2013.

[26] KÖHL, M. A.. **An Executable Structural Operational Formal Semantics for Python**. Master's dissertation, Saarland University, 12 2020.

[27] OHEARN, P. W.. **Resources, concurrency, and local reasoning**. Theor. Comput. Sci., 375, 2007.

[28] LEA, D.. **Concurrent Programming in Java: Design Principles and Patterns**. Addison-Wesley Longman Publishing Co., Inc., USA, 1996.

[29] HANSEN, P. B.. **Monitors and concurrent pascal: a personal history**. SIGPLAN Not., 28(3), 1993.

[30] BACON, D. F.; STROM, R. E. ; TARAFDAR, A.. **Guava: a dialect of java without data races**. SIGPLAN Not., 35(10), 2000.