Carl von Ossietzky Universität Oldenburg

II. School of Computing Science, Business Administration, Economics, and Law
Department of Computing Science

Master's Thesis

Submitted to the Formal Methods Research Group
in Partial Fullfilment of the Requirements for the Degree of

Master of Science

Interval Reasoning for C11 RAR

by

Florian Dyck

Thesis Supervisor:
Prof. Dr. Heike Wehrheim,
Lara Bargmann, M. Sc.

Oldenburg, October 17, 2024

Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Ort, Datum	Unterschrift

Abstract.
Owicki-Gries reasoning extends Hoare logic to concurrent programs under the assumption of *sequential consistency*. In this thesis, we develop a proof calculus using interval reasoning for C11 RAR, a fragment of C11 with relaxed and release-acquire memory operations. Our proof calculus allows all Owicki-Gries proof rules not specific to memory operations to remain unchanged. We use an existing model for the C11 RAR semantics and adapt the assertion language Piccolo to reason on its states. For each thread we overapproximate in which order its thread views might occur in sets of lists, on which we define interval assertions. This way we need only a small number of model-specific assertions, which can be combined to create more complex assertions. We formally prove the soundness of our proof rules and show their utility by verifying a number of C11 litmus tests, and a C11 version of Peterson's algorithm.

Contents

1 Introduction

In this thesis we construct and use a proof calculus for C11 RAR [DDDW20]. We adapt Piccolo's [LDW23] syntax to C11 and define their evaluation on program states. With this, we construct proof rules and use them to verify the correctness of several examples.

In 1969 Hoare established his axiomatic reasoning technique to verify the correctness of sequential programs [Hoa69]. Susan Owicki and David Gries extended it to concurrent programs [OG76] with an additional rule for parallel composition. This, however, works only under the assumption of *sequential consistency* [Lam79].

Memory models which give only weaker guarantees and allow behavior violating sequential consistency are called *weak memory models*, for example SRA [LGV16], C11 RAR [DDDW20], and TSO [OSS09, SSO^{+}10]. These describe behavior of modern programming languages and hardware architectures like C11 and x86 processors.

The C11 Memory Model One such weak memory model was introduced for the C programming language with C11 [ISO11]. Early program logics and their extensions [VN13, TVD14, HVQF16, DV16, DV17, HVQF18, HQF18, HQX20] used separation logic to reason directly on an axiomatic semantics. However, this approach uses a notion of *ownership*, which requires many additional proof rules.

[LV15] used Owicki-Gries reasoning, albeit with a strengthened noninterference check. To improve reasoning and solve problems like *out-of-thin-air reads*, stronger versions of the C11 memory model were introduced, e. g. SRA [LGV16], RA+NA [KDD^{+}17] and RC11 [LVK^{+}17]. The former two are already defined in an operational manner and for RC11 an operational semantics equivalent to its RAR-subset (including releasing, acquiring and relaxed operations) has been developed in [DDWD19]. This allows reasoning about programs with the standard validity definition of Hoare logic. [DDDW20] further develops the RC11 operational semantics and introduces standard Owicki-Gries reasoning for it. We build our proof calculus on this semantics.

$$\begin{array}{c}
\textbf{Init} : \mathtt{x} := 0\,;\, \mathtt{y} := 0; \\
\{t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_0 \wedge \mathbb{1}^{\mathtt{y}}_0)\}
\end{array}$$

$$\begin{array}{l||l}
\textbf{Thread } \mathbf{t_1} & \textbf{Thread } \mathbf{t_2} \\
\{t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_0 \wedge \mathbb{1}^{\mathtt{y}}_0)\} & \{t_2 \ltimes \mathbb{1}^{\mathtt{y}}_0; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\} \\
1 : \mathtt{x} := 1; & 3 : \mathtt{a} \leftarrow^{A} \mathtt{y}; \\
\{t_1 \ltimes ([\mathtt{x} = 1] \wedge \mathbb{1}_{\mathtt{y}}) \wedge t_2 \ltimes \mathbb{1}^{\mathtt{y}}_0\} & \{\mathtt{a} = 1 \Rightarrow t_2 \ltimes [\mathtt{x} = 1]\} \\
2 : \mathtt{y} :=^{R} 1 & 4 : \mathtt{b} \leftarrow \mathtt{x} \\
\{\mathit{true}\} & \{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}
\end{array}$$

$$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$$

Figure 1.1: Message Passing. More details in Section 5.4

Reasoning with Potentials For reasoning on weak memory models, [LDW23] introduced Piccolo, which uses inspirations from *interval logics* like LTL [Mos12] and duration calculus [CHR91] to reason about which values shared variables may assume in which sequence. These sequences of possible future values are called *potentials*. The underlying program semantics is parametric with regards to the memory model and was first instantiated with SRA. First steps to reason over multiple weak memory models at the same time were taken in [BW23]. In the same spirit [BDW24] extends Piccolo to work with both SC and TSO, showing that it is possible and yields useful results. Additionally, this allows techniques like [DLW24] to extend Piccolo with the capability to reason about deadlocks, which then can be used in all instances where the Piccolo framework is employed. In this thesis, we adapt Piccolo to C11 RAR.

Motivating Example We show how Piccolo can be used for C11 in Figure 1.1.

Thread t_1 writes the message 1 to the shared variable x and then writes 1 to y, indicating that the message is written. To ensure proper synchronization, y must be written by a *releasing write* and read by an *acquiring read*, denoted by the R and A in the operations, respectively. After this synchronization occurred, thread t_2 is guaranteed to read the message 1 from x.

In the beginning all threads perceive only one write with value 0 to both x and y, which we describe with $t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_0 \wedge \mathbb{1}^{\mathtt{y}}_0)$. $t \ltimes I$ denotes that thread t perceives some interval assertion I, t_0 is a shortcut to describe the perception of all threads, and $\mathbb{1}^{x}_{e}$ describes that only one write to x with value e is observed.

First, t_1 may write $\mathtt{x} := 1$, after which t_1 must perceive $[\mathtt{x} = 1]$, which is written down as $t_1 \ltimes [\mathtt{x} = 1]$. $\mathbb{1}_{\mathtt{y}}$ describes that there is only one write to y and does not restrict its value. Next, a releasing write $\mathtt{y} :=^{R} 1$ may occur. Afterwards, $t_2 \ltimes \mathbb{1}^{\mathtt{y}}_0; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]$ holds. The first part, $\mathbb{1}^{\mathtt{y}}_0$, describes the time until the new write is perceived by t_2. $[\overline{A}(\mathtt{y})]$ describes state which cannot occur after an acquiring read, which is necessary for synchronization. Finally, $[\mathtt{x} = 1]$ describes the message x, which is transferred in the synchronization process.

Thread t_2 initially perceives $\mathbb{1}^{\mathtt{y}}_0; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]$. After executing $\mathtt{a} \leftarrow^{A} \mathtt{y}$, the assertion $\mathtt{a} = 1 \Rightarrow t_2 \ltimes [\mathtt{x} = 1]$ holds, because $\mathtt{a} = 1$ cannot result from a read in $\mathbb{1}^{\mathtt{y}}_0$, where

$\mathtt{y}$ is 0, and an acquiring read from $\mathtt{y}$ cannot read state described by $[\overline{A}(\mathtt{y})]$. Only the final part $[\mathtt{x} = 1]$ of the interval assertion remains. The assertion $t_2 \ltimes [\mathtt{x} = 1]$ resulting from this describes a state, where the only value that can be read from $\mathtt{x}$ is 1, resulting in the postcondition $\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1$.

Overview Chapter 2 presents the syntax and semantics of C11 RAR [DDDW20]. For this semantics, Chapter 3 adapts assertion syntax of Piccolo [LDW23] and defines the validity of assertions. Chapter 4 introduces our proof rules, which Chapter 5 uses to show the validity of several standard litmus tests and a C11 version of Peterson's algorithm. Chapter 6 puts our results in the context of related and possible future work.

2 Program Syntax and Semantics

This chapter introduces the C11 RAR (release-acquire relaxed) semantics from [DDDW20]. There they are shown to coincide with previous semantics from [DDWD19], which itself coincides with the axiomatic semantics from [LVK$^+$17].

2.1 Program Syntax

The syntax of sequential programs is given in Figure 2.1. It uses the set of local variables Var_L, a set of shared variables Var_G and a set of allowed values Val. We usually use $\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots$ to denote local variables Var_L and $\mathtt{x}, \mathtt{y}, \ldots$ to denote shared variables Var_G. If we refer to generic variables of those types, we instead use r and x, respectively. Let $Var = Var_L \uplus Var_G$ denote the set of all variables. Here $\uplus$ denotes the union of disjoint sets. This avoids confusion and ambiguity regarding which variable is used in an expression.

Arithmetic expressions e_a are built from constant values, local variables and their combinations using arithmetic operators. Boolean expressions e_b are built from constant values, local variables, their combinations using logical operators and comparisons of arithmetic expressions. Expressions e are either arithmetic expressions or boolean expressions.

Statements c may change the state of the program and can read or write shared variables. $x :=^R n$ and $r \leftarrow^A x$ denote a releasing write and acquiring read respectively, while the versions without the R/A annotation denote their relaxed versions. The RA in the $x.\mathbf{swap}(v_{\mathtt{W}})^{\mathrm{RA}}$ operation denotes that it is both an acquiring read and then a releasing write with value $v_{\mathtt{W}}$.

Instrumented statements $\tilde{c}$ extend statements with *auxiliary variables*, which do not affect the meaning of the program. They are required to be used in only one thread each, and only in assignments to auxiliary variables. These assignments happen atomically with the statement before them, denoted by the angled brackets.

$$\begin{array}{lll}
\textit{arithmetic expressions} & e_a ::= & v \mid r \mid e_a + e_a \mid e_a - e_a \mid e_a * e_a \mid e_a \div e_a \mid \dots \\
\textit{boolean expressions} & e_b ::= & v \mid r \mid \neg e_b \mid e_b \wedge e_b \mid e_b \vee e_b \mid \dots \mid \\
 & & e_a = e_a \mid e_a < e_a \mid e_a > e_a \mid \dots \\
\textit{expressions} & e ::= & e_b \mid e_a \\
\textit{statements} & c ::= & \mathbf{skip} \mid x.\mathbf{swap}(v)^{\mathrm{RA}} \mid r := e \mid \\
 & & x := e \mid x :=^R e \mid r \leftarrow x \mid r \leftarrow^A x \\
\textit{instrumented statements} & \tilde{c} ::= & \langle c, aux := e \rangle \\
\textit{compound statements} & C ::= & c \mid \tilde{c} \mid C\,;C \mid \\
 & & \mathbf{if}\, e_b \,\mathbf{then}\, C \,\mathbf{else}\, C \,\mathbf{fi} \mid \mathbf{while}\, e_b \,\mathbf{do}\, C \,\mathbf{od}
\end{array}$$

Figure 2.1: Sequential Program Grammar for $r \in \mathit{Var}_L$, $x \in \mathit{Var}_G$, $v \in \mathit{Val}$ and auxiliary variables $aux \in \mathit{Var}_L$

A sequential program C consists of statements, their sequential compositions, conditionals and loops.

This semantics assumes concurrency at top level only. Therefore the structure of a parallel program is assumed to be **Init** ; $(C_1 \| \dots \| C_n)$. Here $C_1 \| \dots \| C_n$ denotes the parallel composition of the programs $C_1, \dots, C_n$. **Init** initializes the values of the variables, which determines the initial state. The parallel program is represented by a mapping $\mathit{Prog} : \mathit{Tid} \to C$, mapping a set of thread identifiers Tid to their respective sequential programs. We usually use t and t' to refer to a generic thread and denote specific threads with t_i for some integer i to avoid confusion between thread identifiers and constant values.

2.2 Actions

Actions are used in for synchronization between local and global state transitions and as part of the global state. They are defined in Figure 2.2. The formal definitions of the smallest sets containing certain actions are given in the upper half of the figure. In the table below, the composition of the other sets is shown. In the bottom line are representations for what the different actions look like. Any set denoted in an area above an action contains all actions like the ones denoted below. Any set stretching over multiple other sets is their union.

$\mathtt{U} = \{upd^{RA}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \mid x \in \mathit{Var}_G, v_{\mathtt{R}}, v_{\mathtt{W}} \in \mathit{Val}\}$ is the set of all update actions. The graphic visualizes the relations of the different sets. $\mathtt{W}_R = \mathtt{U} \cup \{wr^R(x, v_{\mathtt{W}}) \mid x \in \mathit{Var}_G$, $v_{\mathtt{W}} \in \mathit{Val}\}$ and $\mathtt{R}_A = \mathtt{U} \cup \{rd^A(x, v_{\mathtt{R}}) \mid x \in \mathit{Var}_G, v_{\mathtt{R}} \in \mathit{Val}\}$ are the sets of releasing writes and acquiring reads respectively. Notice that both contain the set of update actions. $\mathtt{W}_X = \{wr(x, v_{\mathtt{W}}) \mid x \in \mathit{Var}_G, v_{\mathtt{W}} \in \mathit{Val}\}$ and $\mathtt{R}_X = \{rd(x, v_{\mathtt{R}}) \mid x \in \mathit{Var}_G, v_{\mathtt{R}} \in \mathit{Val}\}$ are the sets of relaxed writes and reads. $\mathtt{W} = \mathtt{W}_R \cup \mathtt{W}_X$ and $\mathtt{R} = \mathtt{R}_A \cup \mathtt{R}_X$ are the sets of all writes and reads, no matter whether they are releasing, acquiring or relaxed. Finally,

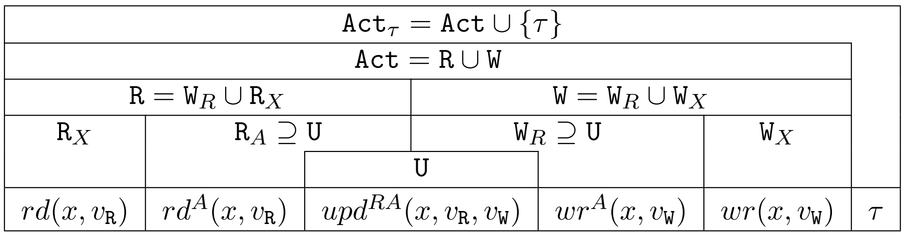

$\mathtt{Act}_\tau = \mathtt{Act} \cup \{\tau\}$
$\mathtt{Act} = \mathtt{R} \cup \mathtt{W}$
$\mathtt{R} = \mathtt{W}_R \cup \mathtt{R}_X$
$\mathtt{W} = \mathtt{W}_R \cup \mathtt{W}_X$
$\mathtt{R}_X$
$\mathtt{R}_A \supseteq \mathtt{U}$
$\mathtt{W}_R \supseteq \mathtt{U}$
$\mathtt{W}_X$
$\mathtt{U}$
$rd(x, v_\mathtt{R})$
$rd^A(x, v_\mathtt{R})$
$upd^{RA}(x, v_\mathtt{R}, v_\mathtt{W})$
$wr^A(x, v_\mathtt{W})$
$wr(x, v_\mathtt{W})$
τ

Figure 2.2: Actions

a	$var(a)$	$rdval(a)$	$wrval(a)$
$rd(x, v_\mathtt{R})$	x	$v_\mathtt{R}$	-
$rd^A(x, v_\mathtt{R})$	x	$v_\mathtt{R}$	-
$wr(x, v_\mathtt{W})$	x	-	$v_\mathtt{W}$
$wr^A(x, v_\mathtt{W})$	x	-	$v_\mathtt{W}$
$upd^{RA}(x, v_\mathtt{R}, v_\mathtt{W})$	x	$v_\mathtt{R}$	$v_\mathtt{W}$

Table 2.1: Accessing the variable, read value and written value of an action a

$\mathtt{Act} = \mathtt{W} \cup \mathtt{R}$ is the set of both reads and writes. $\mathtt{Act}_\tau = \mathtt{Act} \cup \{\tau\}$ adds an additional τ, representing a silent action.

Table 2.1 defines ways to access the variables, read and written values of actions. For these the functions *var*, *rdval* and *wrval* are used respectively.

2.3 Program States

A C11 state $\Gamma \in \Sigma_{C11}$ consists of both a local state $lst \in \Sigma_L$ and a global state $\sigma \in \Sigma_G$. They are defined in Figure 2.3. The local state lst maps for each thread its local variables to their values. $ls : Var_L \nrightarrow Val$ is the local state of a single thread. Let $\nrightarrow$ refer to partial functions, which in this case only map the variables of the thread in question to values. To avoid confusion, let the local states of different threads be disjoint, i. e. $dom(lst(t)) \cap dom(lst(t')) = \emptyset$ if $t \neq t'$. Let $[\![e]\!]_{ls}$ be the evaluation of an expression e in ls. We further use $ls[r := n]$ to represent the local state ls with only the value of r replaced by n.

A global state σ consist of all previous writes, a family of thread views for each thread, of modification views for each write and a set of covered writes. A write is a combination of a write action $w \in \mathtt{W}$ and a timestamp $q \in \mathbb{Q}$. Thread views map each thread to the writes it currently perceives and modification views map each write to the thread view of the writing thread directly after the write. *covered* is a subset of *writes*, specifying which writes were read by an atomic read-modify-write operation.

We lift *var* and *wrval* from write actions to writes ($var((a, q)) = var(a)$ and $wrval((a, q)) = vrwal(a)$) and introduce *tst* to access their timestamp $tst((a, q)) = q$.

$$\begin{aligned}
\Sigma_L &= \{lst \mid lst : Tid \to (Var_L \nrightarrow Val)\} \\
ls[r := n] : \mathtt{a} &\mapsto \begin{cases} n & \text{if } a = r \\ lst(a) & \text{otherwise} \end{cases} \qquad \text{for } ls : Var_L \nrightarrow Val \\
\Sigma_G &= \{(writes, (tview_t)_{t \in Tid}, (mview_w)_{w \in writes}, covered) \mid \\
&\qquad covered \subseteq writes \subseteq \mathtt{W} \times \mathbb{Q}, tview_t, mview_w : Var_G \to writes\}
\end{aligned}$$

Figure 2.3: C11 RAR States

$$\begin{aligned}
\textbf{Init} &= x_1 := k_1 \,;\cdots;\, x_n := k_n \,;\, [r_1 := l_1;] \ldots [r_m := l_m;] \\
lst_{\textbf{Init}}(t) &: r_i \mapsto l_i \quad \text{for each thread } t \text{ and its local variables } r_i \\
writes_{\textbf{Init}} &= \{(wr(x_i, k_i), 0) \mid 1 \leq i \leq n\} \\
view_{\textbf{Init}} &: Var_G \to writes, x_i \mapsto (wr(x_i, k_i), 0) \quad \text{for each } x_i \in Var_G \\
tview_{\textbf{Init}} &= view_{\textbf{Init}} \quad \text{for each thread } t \\
mview_{\textbf{Init}} &= view_{\textbf{Init}} \quad \text{for each } w \in writes_{\textbf{Init}} \\
\sigma_{\textbf{Init}} &= (writes_{\textbf{Init}}, (tview_{\textbf{Init}})_{t \in Tid}, (mview_{\textbf{Init}})_{w \in writes}, \emptyset) \\
\Gamma_{\textbf{Init}} &= (lst_{\textbf{Init}}, \sigma_{\textbf{Init}})
\end{aligned}$$

Figure 2.4: Initial State

2.4 Initial State

The initial state is defined in Figure 2.4. It is determined by **Init**, which initializes a value to each shared variable and any number of local variables. The local state of each thread maps its local variables to their initial values written in **Init**. $writes_{\textbf{Init}}$ contains a write with each shared variable's initial value and timestamp 0. The initial thread and modification views all map each shared variable to its initial write. $\sigma_{\textbf{Init}}$ is composed of $writes_{\textbf{Init}}$, $tview_t$ for each thread t, $mview_{\textbf{Init}}$ for each initial write w and an empty set *covered*. Finally, $\Gamma_{\textbf{Init}}$ is defined as the combination of the local and shared initial state.

2.5 Program Semantics

The transition relation $\Longrightarrow$ on C11 states are defined by a combination of transitions on the local and global state. They synchronize using the set of actions $\mathtt{Act}_\tau$:

$$\frac{(P, lst) \xrightarrow{\tau}_t (P', lst')}{(P, lst, \sigma) \Longrightarrow (P', lst', \sigma)} \qquad \frac{(P, lst) \xrightarrow{a}_t (P', lst') \quad \sigma \overset{a}{\rightsquigarrow}_t \sigma'}{(P, lst, \sigma) \Longrightarrow (P', lst', \sigma')}$$

A synchronized transition step is possible if and only if there are a local transition

$$\frac{r \in \mathit{Var}_L \qquad n = [\![e]\!]_{ls}}{(r := e, ls) \xrightarrow{\tau} (\mathbf{skip}, ls[r := n])} \qquad \frac{x \in \mathit{Var}_G \qquad a = wr^{[R]}(x, [\![e]\!]_{ls})}{(x :=^{[R]} e, ls) \xrightarrow{a} (\mathbf{skip}, ls)}$$

$$\frac{a = rd^{[A]}(x,n) \qquad n \in \mathit{Val}}{(r \leftarrow^{[A]} x, ls) \xrightarrow{a} (\mathbf{skip}, ls[r := n])} \qquad \frac{a = upd^{RA}(x,m,n) \qquad m \in \mathit{Val}}{(x.\mathbf{swap}(n)^{\mathrm{RA}}, ls) \xrightarrow{a} (\mathbf{skip}, ls)}$$

$$\frac{(C_1, ls) \xrightarrow{a} (C_1', ls')}{(C_1 \,;\, C_2, ls) \xrightarrow{a} (C_1' \,;\, C_2, ls')} \qquad \frac{}{(\mathbf{skip} \,;\, C_2, ls) \xrightarrow{\tau} (C_2, ls)}$$

$$\frac{[\![e_b]\!]_{ls}}{(\mathbf{if}\, e_b\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2\, \mathbf{fi}, ls) \xrightarrow{\tau} (C_1, ls)} \qquad \frac{\neg[\![e_b]\!]_{ls}}{(\mathbf{if}\, e_b\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2\, \mathbf{fi}, ls) \xrightarrow{\tau} (C_2, ls)}$$

$$\frac{[\![e_b]\!]_{ls} \qquad \mathit{WHILE} = \mathbf{while}\, e_b\, \mathbf{do}\, C\, \mathbf{od}}{(\mathit{WHILE}, ls) \xrightarrow{\tau} (C \,;\, \mathit{WHILE}, ls)} \qquad \frac{\neg[\![e_b]\!]_{ls}}{(\mathbf{while}\, e_b\, \mathbf{do}\, C\, \mathbf{od}, ls) \xrightarrow{\tau} (\mathbf{skip}, ls)}$$

$$\textsc{Aux}\ \frac{\begin{array}{c}(c, ls) \xrightarrow{a} (\mathbf{skip}, ls') \\ (\mathtt{a} := e, ls' \xrightarrow{\tau} (\mathbf{skip}, ls''))\end{array}}{(\langle c, \mathtt{a} := e\rangle, ls) \xrightarrow{a} (\mathbf{skip}, ls'')} \qquad \textsc{Prog}\ \frac{(P(t), lst(t)) \xrightarrow{a} (C, ls) \quad a \in \mathtt{Act}_\tau}{(P, lst) \xrightarrow{a}_t (P[t := C], lst[t := ls])}$$

Figure 2.5: Local Transitions

step $\xrightarrow{a}$ on the local state and a memory transition step $\overset{a}{\rightsquigarrow}_t$ on the global state with the same action a. Transitions with the silent action τ do not need to be synchronized. There is only a rule for local transitions with the silent action because in C11 RAR there are no memory transitions with the silent action.

This definition allows for a generic local transition relation which can be used for different memory models. The memory transition relation on the other hand is specific to the memory model, in this case C11 RAR.

2.5.1 Local Program Semantics

The local transition relation is defined in Figure 2.5. An assignment assigns the value of an expression e in the local state of a thread. Memory transition steps update the local state as necessary and use the annotated action to synchronize with matching memory transition steps. The other transition steps are as expected. Transition steps are lifted from a single statement to a sequence of statements. An if statement executes the first subprogram if the condition is true, otherwise the second subprogram. A while loop executes its body and itself again if the condition is true, otherwise it is skipped. Assignments to auxiliary variables atomically change the local state additionally to whatever the original statement was doing. Finally, Prog lifts sequential programs to parallel programs.

2.5.2 Memory Semantics

In this section, we introduce the memory transition relation $\rightsquigarrow$ for C11. This is the part of the semantics specific to C11. We begin by introducing the definitions which are used in the transition rules.

For new writes a new timestamp must be established, such that it occurs after the writes the writing thread read. $\sigma.\mathit{fresh}$ gives the opportunity to define a timestamp directly after another write:

$$\sigma.\mathit{fresh}(q, q') \equiv q < q' \wedge (\forall w' \in \sigma.\mathit{writes} : q < \mathit{tst}(w') \implies q' < w') \tag{2.1}$$

The writes to a shared variable are strictly ordered by their timestamps. To achieve this, a thread must only perceive writes which are the current write or occur after this write. For this, the set of *observable writes* defines which writes to a variable a thread may perceive:

$$\sigma.OW(t, x) := \{(a, q) \in \sigma.\mathit{writes} \mid \mathit{var}(a) = x \wedge \mathit{tst}(\sigma.\mathit{tview}_t(x)) \leq q\} \tag{2.2}$$

When synchronizing with releasing writes and acquiring reads, we update the thread view of the reading thread. For this, we take the newer timestamp for each shared variable from the thread view and modification view of the write which is read from. This is done using the function $\otimes$:

$$(\delta_1 \otimes \delta_2)(x) := \begin{cases} \delta_1(x) & \text{if } \mathit{tst}(\delta_2(x)) \leq \mathit{tst}(\delta_1(x)) \\ \delta_2(x) & \text{otherwise} \end{cases} \tag{2.3}$$

Using these functions, Figure 2.6 defines the memory transition relation.

Read chooses an observable write (w, q) to read from. If the write is releasing and the read is acquiring, the view of the reading thread is updated to be at least as new for each shared variable as the modification view of that write, using $\otimes$. Otherwise, only the read variable is updated to the read write.

Write chooses an observable write (w, q) to write after, which is not already covered by an update. It updates the thread view of the written variable to the write and copies the current thread view to the modification view of the write. The write is also added to the set *writes* of the global state σ.

Update chooses an observable write (w, q) the same way as Write. It has the same effect as first reading a value acquiringly and then writing directly after the read. Additionally, the write is added to the set *covered*, preventing further writes (and updates) directly after this write.

Example 2.1. *An example can be seen in Figure 2.7. The figure on the left shows views from a state σ after statements 1, 2 and 3 of the program on the right are executed in this order and a state σ' after additionally statement 4 is executed. In this case, tview_{t_1}*

Read

$$\frac{\begin{array}{c} a \in \{rd(x, v_{\mathtt{R}}), rd^{A}(x, v_{\mathtt{R}})\} \qquad (w, q) \in \sigma.OW(t, x) \\ wrval(w) = v_{\mathtt{R}} \qquad tview'_t = \begin{cases} \sigma.tview_t \otimes \sigma.mview_{(w,q)} & \text{if } (w, a) \in \mathtt{W}_R \times \mathtt{R}_A \\ \sigma.tview_t[x := (w, q)] & \text{otherwise} \end{cases} \end{array}}{\sigma \overset{a}{\rightsquigarrow}_t \sigma[tview_t := tview'_t]}$$

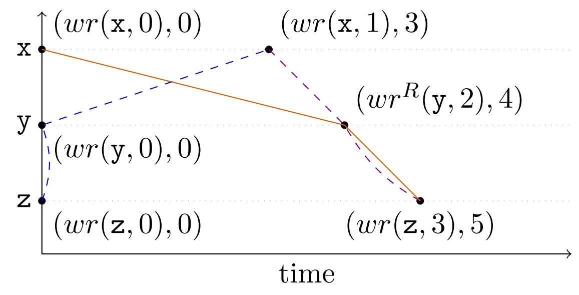
Write

$$\frac{\begin{array}{c} a \in \{wr(x, v_{\mathtt{W}}), wr^{R}(x, v_{\mathtt{W}})\} \qquad (w, q) \in \sigma.OW(t, x) \backslash \sigma.covered \\ \sigma.fresh(q, q') \qquad writes' = \sigma.writes \cup \{(a, q')\} \qquad tview'_t = tview_t[x := (a, q')] \end{array}}{\sigma \overset{a}{\rightsquigarrow}_t \sigma[tview_t := tview'_t, mview_{(a,q')} := tview'_t, writes := writes']}$$

Update

$$\frac{\begin{array}{c} a = upd^{RA}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \qquad (w, q) \in \sigma.OW(t, x) \backslash \sigma.covered \qquad wrval(w) = v_{\mathtt{R}} \\ \sigma.fresh(q, q') \qquad writes' = \sigma.writes \cup \{(a, q')\} \qquad covered' = \sigma.covered \cup \{(a, q)\} \\ tview'_t = \begin{cases} \sigma.tview_t[x := (a, q')] \otimes \sigma.mview_{(w,q)} & \text{if } w \in \mathtt{W}_R \\ \sigma.tview_t[x := (a, q')] & \text{otherwise} \end{cases} \end{array}}{\begin{array}{c} \sigma \overset{a}{\rightsquigarrow}_t \sigma[tview_t := tview'_t, mview_{(a,q')} := tview'_t, \\ writes := writes', covered = covered'] \end{array}}$$

Figure 2.6: Memory Transitions



Init : $\mathtt{x} := 0$; $\mathtt{y} := 0$; $\mathtt{z} := 0$;

Thread $\mathtt{t_1}$	**Thread $\mathtt{t_2}$**
$1 : \mathtt{z} := 3;$	$3 : \mathtt{x} := 3;$
$2 : \mathtt{y} :=^R 2$	$4 : \mathtt{a} \leftarrow^A \mathtt{y}$

Figure 2.7: Illustration of views and their updates: blue: thread view before $\mathtt{a} \leftarrow^A \mathtt{y}$, orange: $mview_{(wr^R(\mathtt{y},2),4)}$, violet: thread view after acquiring read on $(wr^R(\mathtt{y}, 2), 4)$

and $mview_{(wr^R(\mathtt{y},2),4)}$ are the orange thread views of t_1 and modification views of the write $(wr^R(\mathtt{y},2),4)$ both before and after the transition step, in both σ and σ'. $\sigma.tview_{t_2}$ is the blue thread view of t_2 before and $\sigma'.tview_{t_2}$ is the violet thread view of t_2 after the transition step.

In the beginning there are initial writes to `x`, `y` *and* `z` *with the value 0 at the timestamp 0. By executing the statements in the order 1, 2 and finally 3, the resulting writes have increasingly lower timestamps because the fresh relation inserts new writes directly after another write to the same variable.*

The $tview_{t_1}$ is created by executing statements 1 and 2, and by virtue of being $tview_{t_1}$ after statement 2 also $tview_{(wr^R(\mathtt{y},2),4)}$. $\sigma.tview_{t_2}$ simply results by executing statement 3, only changing the value of the written variable. In this case executing statement 4 and reading $(wr^R(\mathtt{y},2),4)$ not only updates the value of the view of `y`*, but also each other value to the newer one of the corresponding memory view and the current thread view. This way only the value for* `x` *is updated, resulting in $\sigma'.tview_{t_2}$.*

2.6 Well Formedness

Every global state reachable with a finite computation fulfills certain conditions, which we summarize as it being well-formed, as defined below:

$$\begin{aligned} wfs(\sigma) \equiv\ & ran((\bigcup_t \sigma.tview_t) \cup (\bigcup_w \sigma.mview_w)) \subseteq \sigma.writes \wedge \\ & finite(\sigma.writes) \wedge \sigma.covered \subseteq \sigma.writes \wedge \\ & (\forall w : w \in \sigma.writes \Rightarrow \sigma.mview_w(var(w)) = w) \end{aligned}$$

Lemma 2.2. *Every reachable global state σ is well-formed, i. e. $wfs(\sigma)$ holds.*

This is proven by induction in Appendix B.1.

Because only such states are reachable, from now on we only consider *well-formed* global states. In the next chapter we continue by defining our assertions on C11 states (with such well-formed global states).

3 Assertions

In this chapter, we introduce assertions, which we use to describe C11 states. The syntax for assertions, interval assertions and extended expressions is defined in Figure 3.1.

Extended expressions are evaluated on views (and C11 states), which map shared variables to writes, identical to thread and modification views. In comparison to *expressions* (see Figure 2.1) which only contain local variables, they may also include shared variables. They make assertions at one possible point in time, for a specific thread view of a thread.

Interval assertions hold for lists of views (and C11 states), which overapproximate in which order views can be thread views of a thread. For both of these, the C11 states are necessary to provide information not contained in the lists, such as values of local variables and modification views. They make assertions for all possible sequences of thread views in which thread views of some thread might occur.

Assertions hold for C11 states. They can make assertions for every thread by containing interval assertions and the information for which thread these interval assertions hold.

$$
\begin{array}{lrl}
\textit{extended arithmetic expressions} & E_a ::= & e_a \mid x \mid \\
 & & E_a + E_a \mid E_a - E_a \mid E_a * E_a \mid \dots \\
\textit{extended boolean expressions} & E_b ::= & e_b \mid x \mid \overline{A}(x) \mid \\
 & & E_a = E_a \mid E_a \geq E_a \mid \dots \mid \\
 & & \neg E_b \mid E_b \wedge E_b \mid \dots \\
\textit{interval assertions} & I ::= & [E_b] \mid \mathbb{1}_x \mid C_x \mid I;I \mid I \wedge I \mid I \vee I \\
\textit{assertions} & \varphi, \psi ::= & t \ltimes I \mid e_b \mid \varphi \wedge \varphi \mid \varphi \vee \varphi
\end{array}
$$

Figure 3.1: Assertion Syntax

3.1 Views

We begin by defining views and the precedence relation. A view is a mapping of shared variables to writes, the same way as thread views and modification views. These mappings are only restricted in that shared variables must be mapped to writes of the same variable.

$$\sigma.\mathit{Views} := \{\delta : \mathit{Var}_G \to \sigma.\mathit{writes} \mid \forall x \in \mathit{Var}_G : \mathit{var}(\delta(x)) = x\} \tag{3.1}$$

Further we define the precedence/ succession relation, which describes in which orderings thread views might occur. A write w precedes another write w' ($w \preceq w'$) if and only if $\mathit{tst}(w) \leq \mathit{tst}(w')$. We alternatively write w' succeeds w ($w' \succeq w$), which has the same meaning as $w \preceq w'$. We lift these definitions to views, such that a view δ precedes δ' and δ succeeds δ' if and only if for each shared variable x, $\delta(x)$ precedes $\delta'(x)$:

$$\delta \preceq \delta' \equiv \delta' \succeq \delta \equiv \forall x \in \mathit{Var}_G : \delta(x) \preceq \delta'(x) \tag{3.2}$$

The semantic of our assertions is built on the principle that the succession relation overapproximates in which order thread views can occur. Theorem 3.1 states that, if a thread observes first the view δ and then δ', then $\delta \preceq \delta'$ must hold. This means that if $\delta \preceq \delta'$ does not hold, δ' cannot occur as thread view of some thread after δ.

Theorem 3.1. *For any thread t and any states $\Gamma = (\mathit{lst}, \sigma)$ and $\Gamma' = (\mathit{lst}', \sigma')$ with $(P, \Gamma) \Longrightarrow (P', \Gamma')$, the thread view of t cannot decrease: $\sigma.\mathit{tview}_t \preceq \sigma'.\mathit{tview}_t$*

Proof. The updates in the memory transitions only allow changing the value of the thread t view in the following ways.

- The thread view may be updated using $\otimes$. This new thread view must succeed the old one because for each shared variable $\otimes$ maps to the write with the greater timestamp by definition of $\otimes$ (Equation (2.3)).
- Updating the value for a single shared variable x. Choosing an observable write $(a, q) \in \sigma.\mathit{OW}(\mathit{tst}(\sigma.\mathit{tview}_t(x)), x)$ lets us conclude $\mathit{tst}(\sigma.\mathit{tview}_t(x)) \leq q$ by definition of OW (Equation (2.2)). In case of a read, the new write is updated to this q. In case of a write, the new write is also updated to some q', which is greater than q by definition of *fresh* (Equation (2.1)).

Other changes to the thread view are not possible. Thus we can conclude $\sigma.\mathit{tview}_t(x) \preceq \sigma'.\mathit{tview}_t(x)$. □

3.2 Extended Expressions

With an understanding of precedence, we define the evaluation of *extended expressions*. These are used in *interval assertions* and *assertions*.

Definition 3.2. *For a C11 state $\Gamma = (\sigma, lst)$ and view $\delta \in \sigma.Views$, we define the evaluation of extended expressions:*

$$\begin{aligned}
[\![\cdot]\!]_{\Gamma,\delta} &: E_a \cup E_b \to Val \\
[\![r]\!]_{\Gamma,\delta} &= n \textit{ iff } \exists t \in Tid : lst(t)(r) = n \\
[\![x]\!]_{\Gamma,\delta} &= val(\delta(x)) \\
[\![\neg E]\!]_{\Gamma,\delta} &= \neg\, [\![E]\!]_{\Gamma,\delta} \\
[\![E_1 \oplus E_2]\!]_{\Gamma,\delta} &= [\![E_1]\!]_{\Gamma,\delta} \oplus [\![E_2]\!]_{\Gamma,\delta} \\
&\quad \textit{for } \oplus \in \{+, -, *, \div, \dots, =, \leq, \geq, \dots, \wedge, \vee, \dots\} \\
[\![\overline{A}(x)]\!]_{\Gamma,\delta} &= \begin{cases} true, & \textit{if } \delta(x) \in \mathtt{W}_R \times \mathbb{Q} \wedge \neg(\sigma.mview_{\delta(x)} \preceq \delta) \\ false, & \textit{otherwise} \end{cases}
\end{aligned}$$

Local variables $\mathtt{a}$ are evaluated to the value of the local state of that variable $lst(t)(\mathtt{a})$ of the thread t which uses them, because they are only defined for that thread. Shared variables x are evaluated to the value of the view of that variable $val(\delta(x))$.

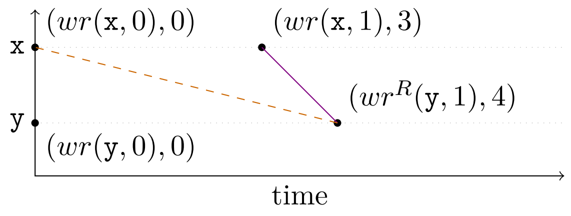
Example 3.3. *Let $\Gamma = (lst, \sigma)$ be a C11 state with $lst(t)(\mathtt{a}) = 1$ for some thread t and $\delta = \{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1)\}$. Then both $[\![\mathtt{a}]\!]_{\Gamma,\delta}$ and $[\![\mathtt{x}]\!]_{\Gamma,\delta}$ evaluate to 1 and consequently $[\![\mathtt{a} = \mathtt{x}]\!]_{\Gamma,\delta}$ evaluates to true.*

$\overline{A}(x)$ is used for releasing writes and acquiring reads. It describes a view which cannot occur after an acquiring read from x, as stated in Lemma 3.4. $\delta(x) \in \mathtt{W}_R \times \mathbb{Q}$ describes that $\delta(x)$ must be a releasing write and $\neg(\sigma.mview_{\delta(x)} \preceq \delta)$ describes that the view δ cannot occur after the modification view $\sigma.mview_{\delta(x)}$ of the write $\delta(x)$ which would be read by such a transition. This way we know if $\overline{A}(x)$ evaluates to true for $\sigma.tview_t$, then $\sigma'.tview_t$ must be a different view, which we later use for our proof rule Ld-a-shift for acquiring reads.

Lemma 3.4. *For any thread t and any global states σ and σ' with $\sigma \overset{rd^A(x,v)}{\rightsquigarrow_t} \sigma'$ and any local state lst', $[\![\overline{A}(x)]\!]_{(lst',\sigma'),\sigma'.tview_t} = false$.*

Proof. Proof by contradiction. Let $\delta = \sigma'.tview_t$. Given the action $a = rd^A(x, v)$, note that $\delta(x) \in \{wr^R(x, v)\} \times \mathbb{Q}$, because this is the write w the thread reads in Read. For the contradiction assume $\delta(x) \in \mathtt{W}_R \times \mathbb{Q} \wedge \neg(\sigma'.mview_{\delta(x)} \preceq \delta)$. By Read, $\delta = \sigma.tview_t \otimes \sigma.mview_{\delta(x)}$. By definition of $\otimes$ (Equation (2.3)), it maps to a view which maps each shared variable $\mathtt{y}$ to the write of $\sigma.tview_t(\mathtt{y})$ and $\sigma.mview_w(\mathtt{y})$ with the greater timestamp. We can conclude $\sigma.mview_{\delta(x)} \preceq \delta$. Further $\sigma.mview_{\delta(x)} = \sigma'.mview_{\delta(x)}$ holds because modification views cannot be changed, only added for new writes. With this $\sigma'.mview_{\delta(x)} \preceq \delta$ must hold, which contradicts our assumption. □

Note that $\neg(\delta \preceq \delta')$ and $\delta' \preceq \delta$ are *not* identical. For example the views $\{x \mapsto (wr(x, 0), 0), \mathtt{y} \mapsto (wr(\mathtt{y}, 1), 1)\}$ and $\{x \mapsto (wr(\mathtt{x}, 1), 1), \mathtt{y} \mapsto (wr(\mathtt{y}, 0), 0)\}$ do fulfill the



Init : $\mathtt{x} := 0\,;\mathtt{y} := 0;$

Thread $\mathbf{t_1}$	**Thread $\mathbf{t_2}$**
$1 : \mathtt{x} := 3;$	$3 : \mathtt{a} \leftarrow^A \mathtt{y};$
$2 : \mathtt{y} :=^R 2$	$4 : \mathtt{b} \leftarrow \mathtt{x}$

Figure 3.2: Illustration for Example 3.5 where $[\![\overline{A}(\mathtt{y})]\!]_{\Gamma,\delta} = \mathit{true}$: violet: $\sigma.\mathit{mview}_{(wr^R(\mathtt{y},1),4)}$ after statement 2, orange: view δ

former, but not the latter. The former indicates that views cannot occur in some order, the latter that they can occur in the reverse order.

Example 3.5. *In this example, we construct a state and view with* $[\![\overline{A}(\mathtt{y})]\!]_{\Gamma,\delta} = \mathit{true}$*. Let* $\Gamma = (\mathit{lst},\sigma)$ *be a C11 state with* $\sigma.\mathit{mview}_{(wr^R(\mathtt{y},1),4)} = \{\mathtt{x} \mapsto (wr(\mathtt{x},1),3), \mathtt{y} \mapsto (wr^R(\mathtt{y},1),4)\}$ *and* δ *be a view with* $\delta = \{\mathtt{x} \mapsto (wr^R(\mathtt{x},0),0), \mathtt{y} \mapsto (wr(\mathtt{y},1),4)\}$*. These views are illustrated in Figure 3.2. On the right side of the figure the Message Passing program is shown, where said state could occur after executing statements 1 and 2. Then* $[\![\overline{A}(\mathtt{y})]\!]_{\Gamma,\delta}$ *evaluates to true because* $\delta(\mathtt{y})$ *is a releasing write and the corresponding modification view* $\sigma.\mathit{mview}_{(wr^R(\mathtt{y},1),4)}$ *does not precede* δ *because the write* $\sigma.\mathit{mview}_{(wr^R(\mathtt{y},1),4)}(\mathtt{x}) = (wr(\mathtt{x},1),3)$ *occurs at a later timestamp than* $\delta(\mathtt{x}) = (wr(\mathtt{x},0),0)$*.*

As stated in Lemma 3.4 this means that δ *cannot occur after an acquiring read from* y*, because the write transition step would use* $\otimes$ *to update the thread view, resulting in violet:* $\sigma.\mathit{mview}_{(wr^R(\mathtt{y},1),4)}$*. From there on the thread can never perceive* $\delta(\mathtt{x}) = (wr(\mathtt{x},0),0)$ *again, because the timestamps of perceived writes must not decrease.*

3.3 Interval Assertions

For *interval assertions*, we are interested in *lists* of views. Such a list overapproximates a possible sequence of thread views for some thread.

A list $L = \langle\delta_1,\ldots,\delta_n\rangle$ is a sequence of elements δ_i. Let $L(i) := \delta_i$ retrieve the ith element of L and $|L| := n$ be the length of the list. With this, for $1 \le i \le j \le |L|$, let $L|_i^j := \langle\delta_i,\ldots,\delta_j\rangle$ be the sublist of L from i to j. We say L contains δ (written $\delta \in L$) if there exists a $1 \le i \le |L|$ with $L(i) = \delta$. Additionally, we use $\cdot$ to concatenate lists: $L_1 \cdot L_2 := \langle L_1(1),\ldots,L_1(|L_1|),L_2(1),\ldots,L_2(|L_2|)\rangle$.

For a list of views specifically, let $L.\mathit{writes} \equiv \bigcup_{\delta\in L} \mathit{ran}(\delta)$ be the set of all writes viewed by some view $\delta \in L$. Here ran refers to the range of a function, in this specific case the writes mapped to by that view.

A list of views can be understood as a possible order in which a thread might perceive views. Later we restrict these lists in their construction such that each view in the list succeeds each previous entry in the list and the thread view of some thread. This overapproximates in which order they might occur as thread views, as stated by

Theorem 3.1. We define the validity of interval assertions for such a list, which specifies a possible ordering of thread views, and a C11 state, which contains further information like values of local variables and modification views.

Definition 3.6. *For a C11 state $\Gamma = (\sigma, lst)$ and a list $L = \langle \delta_1, \ldots, \delta_n \rangle \in \sigma.Views^*$ of views we define the validity of an interval assertion:*

$$\begin{array}{ll}
\Gamma, L \vDash [E_b] & \textit{iff } \forall \delta \in L : [\![E_b]\!]_{\Gamma,\delta} = true \\
\Gamma, L \vDash C_x & \textit{iff } \forall \delta \in L : \delta(x) \in \sigma.covered \\
\Gamma, L \vDash \mathbb{1}_x & \textit{iff } \forall \delta, \delta' \in L : \delta(x) = \delta'(x) \\
\Gamma, L \vDash I_1; I_2 & \textit{iff } \Gamma, L_1 \vDash I_1 \textit{ and } \Gamma, L_2 \vDash I_2 \textit{ for some (possibly empty)} \\
 & \quad L_1 \textit{ and } L_2 \textit{ with } L_1 \cdot L_2 = L \\
\Gamma, L \vDash I_1 \wedge I_2 & \textit{iff } \Gamma, L \vDash I_1 \textit{ and } \Gamma, L \vDash I_2 \textit{ (similarly for } \vee)
\end{array}$$

For extended boolean expressions $[E_b]$ to hold on a list, E_b must evaluate to true for all views in that list. C_x describes that all writes to x in views in the list are in the *covered* set. This being part of interval assertions prevents its negation, which makes our proof rules sound for the swap operation.

$\mathbb{1}_x$ describes that there is only one unique write to x in the list. This is useful to reason on possible behaviors when a new write (to x) is introduced. Such a new write then can only precede or succeed this single write. On the other hand, if there are multiple writes to x in a list, the new write could succeed some writes and precede others.

The chop operator ; requires the existence of some split of the list, such that the first interval assertion is fulfilled by the first sublist and the second interval assertion is fulfilled by the second sublist. This is how we reason about sequence of events.

Finally, boolean operations $\vee$ and $\wedge$ are defined as usual, but negations are not allowed. This allows us to derive Lemma 3.7, stating that if a formula holds on a list, that it also holds on any sublist (including the empty list). This allows us for a given list to derive that all its sublists fulfill interval assertions that the original list fulfills, which is essential for many of our proofs.

If we allowed negation with the usual definition Lemma 3.7 would not hold, a counterexample is $\Gamma, \langle \delta \rangle \vDash \neg[false]$ and $\Gamma, \langle\rangle \nvDash \neg[false]$ for an arbitrary C11 state Γ and view δ.

Lemma 3.7. *For a C11 state Γ and a list L of views, when $\Gamma, L \vDash I$, then $\Gamma, L|_i^j \vDash I$ for every $0 \leq i - 1 \leq j \leq n$ (This may be the empty list for $i = j + 1$).*

For the proof see Appendix B.2.

Example 3.8. *Let Γ be a C11 state and $L = \langle \{\mathtt{x} \mapsto (wr(\mathtt{x}, 0), 0)\}, \{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1)\}, \{\mathtt{x} \mapsto (wr(\mathtt{x}, 2), 2)\} \rangle$ be a list of views. Then $\Gamma, L|_1^1 \vDash \mathbb{1}_{\mathtt{x}}$ holds because there is only one*

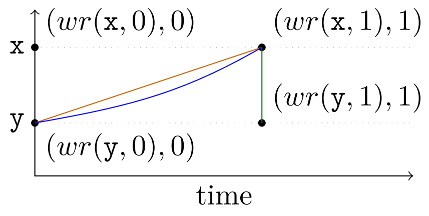


Figure 3.3: $\langle\delta_1, \delta_2\rangle \in \sigma.\mathit{Lists}(t)$: orange: $\sigma.\mathit{tview}_t$, blue: δ_1, green: δ_2

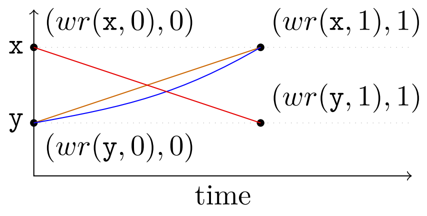


Figure 3.4: $\langle\delta_1, \delta_2'\rangle \notin \sigma.\mathit{Lists}(t)$: orange: $\sigma.\mathit{tview}_t$, blue: δ_1, red: δ_2'

unique write to x *in* $L|_1^1$ *and* $\Gamma, L|_2^3 \vDash [\mathtt{x} > 1]$ *holds because the extended expressions evaluate to true on both views in* $L|_2^3$. *Consequently* $\Gamma, L \vDash \mathbb{1}_{\mathtt{x}}; [\mathtt{x} > 1]$ *holds because* $L = L|_1^1 \cdot L|_2^3$ *and these lists fulfill the interval assertions respectively.*

3.4 Assertions

In a global state σ, we are interested in lists which could reasonably represent an order in which a thread might perceive views. We restrict lists L in the following three ways:

1. Every $L(i)$ has to be in $\sigma.\mathit{Views}$, i. e. $\forall 1 \le i \le |L| : L(i) \in \sigma.\mathit{Views}$
2. Every view $L(i+1)$ should be able to occur after $L(i)$, so we require that $L(i+1)$ succeeds $L(i)$, i. e. $\forall 1 \le i < |L| : L(i) \preceq L(i+1)$
3. The views of the list should be able to occur after the current (thread) view δ and thus must succeed it, i. e. $\forall 1 \le i \le |L| : \delta \preceq L(i)$

This is equivalent to the following definition:

$$\sigma.\mathit{Lists}(\delta) := \{\langle\delta_1, \ldots, \delta_n\rangle \in \sigma.\mathit{Views}^* \mid \forall 1 \le i \le j \le n : \delta \preceq \delta_i \preceq \delta_j\} \tag{3.3}$$

Most of the time we use this as $\sigma.\mathit{Lists}(\sigma.\mathit{tview}_t)$ for a global state σ and a thread t. Therefore, we introduce the shortcut below if the argument is a thread instead of a view:

$$\sigma.\mathit{Lists}(t) := \sigma.\mathit{Lists}(\sigma.\mathit{tview}_t) \tag{3.4}$$

Lemma 3.9 supports that Equation (3.3) describes the criteria above. Further, the right side is simpler and thus less work to show that it holds for a given list. We use this equivalence in proofs to show that certain lists are in $\sigma.\mathit{Lists}(\delta)$.

Lemma 3.9. *For any C11 state* σ *and* $\langle\delta_1, \ldots, \delta_n\rangle \in \sigma.\mathit{Views}^*$ *holds*

$$\forall 1 \le i \le j \le n : \delta \preceq \delta_i \preceq \delta_j \iff (n = 0 \lor \delta \preceq \delta_1) \land \forall 1 \le i < n : \delta_i \preceq \delta_{i+1}$$

The proof is given in Appendix B.3

Example 3.10. *Let* $\Gamma = (lst, \sigma)$ *be a C11 state with* $\sigma.tview_t = \{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1), \mathtt{y} \mapsto (wr(\mathtt{y}, 0), 0)\}$ *and containing the used writes in* $\sigma.writes$ *(implying that all following views are in* $\sigma.Views$*).*

With the views from Figure 3.3, $\langle \delta_1, \delta_2 \rangle = \langle \{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1), \mathtt{y} \mapsto (wr(\mathtt{y}, 0), 0)\}, \{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1), \mathtt{y} \mapsto (wr(\mathtt{y}, 1), 1)\}\rangle$ *is a list in* $\sigma.Lists(t)$*. In this example* $\sigma.tview_t \preceq \delta_1 \preceq \delta_2$ *holds and thus* $\langle \delta_1, \delta_2 \rangle \in \sigma.Lists(t)$*. Note that the empty list* $\langle\rangle$ *is in* $\sigma.Lists(\delta)$ *for any global state* σ *and view* δ*.*

With the views from Figure 3.4, $\langle \delta_1, \delta_2' \rangle = \langle \{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1), \mathtt{y} \mapsto (wr(\mathtt{y}, 0), 0)\}, \{\mathtt{x} \mapsto (wr(\mathtt{x}, 0), 0), \mathtt{y} \mapsto (wr(\mathtt{y}, 1), 1)\}\rangle$ *is in not in* $\sigma.Lists(\delta)$ *for any global state* σ *and view* δ*, because the timestamp for* $\mathtt{x}$ *decreases:* $\delta_1(\mathtt{x}) \npreceq \delta_2'(\mathtt{x})$*. Additionally for this thread view* $\sigma.tview_t(\mathtt{x}) \npreceq \delta_2'(\mathtt{x})$ *also prevents the list from being in* $\sigma.Lists(t)$*.*

In the definition of validity for *assertions*, we use $\sigma.Lists(t)$ to overapproximate in which order thread views may occur.

Definition 3.11. *For a C11 state* $\Gamma = (\sigma, lst)$ *we define the validity of assertion:*

$$\begin{aligned}
&\Gamma \vDash t \ltimes I && \textit{iff } \Gamma, L \vDash I \textit{ for every } L \in \sigma.Lists(tview_t)\\
&\Gamma \vDash e && \textit{iff } \exists \delta \in \sigma.Views : [\![e]\!]_{\Gamma,\delta} = true\\
&\Gamma \vDash I_1 \wedge I_2 && \textit{iff } \Gamma, L \vDash I_1 \textit{ and } \Gamma, L \vDash I_2 \textit{ (similarly for } \vee\textit{)}
\end{aligned}$$

$t \ltimes I$ holds in $\Gamma = (\sigma, lst)$ if I holds for all lists describing some order in which the thread could perceive views, as described by $\sigma.Lists(t)$. Additionally, we introduce the shortcut $t_0 \ltimes I \equiv \bigwedge_{t \in Tid} t \ltimes I$ to assert that all threads must perceive I.

For brevity and to avoid confusion we evaluate expressions the same way as in *extended expressions* (Definition 3.2), except for an arbitrary global state δ. This is possible because their evaluation is independent of δ, since expressions do not contain shared variables.

Finally, boolean operations $\vee$ and $\wedge$ are defined as usual and negations are not allowed for the same reasons as with interval assertions.

Example 3.12. *Let* $\Gamma = (lst, \sigma)$ *be a C11 state with* $\sigma.tview_t = \{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1)\}$ *and* $\sigma.writes = \{(wr(\mathtt{x}, 0), 0), (wr(\mathtt{x}, 1), 1), (wr(\mathtt{x}, 2), 2)\}$*. Then every* $L \in \sigma.Lists(t)$ *consists of lists only containing* $\{\mathtt{x} \mapsto (wr(\mathtt{x}, 1), 1)\}$ *and* $\{\mathtt{x} \mapsto (wr(\mathtt{x}, 2), 2)\}$ *and only in this order. Then* $\Gamma, L|_1^k \vDash \mathbb{1}_{\mathtt{x}}$ *holds for some first part of* L *with* $(wr(\mathtt{x}, 1), 1)$*. For the second part* $\Gamma, L|_{k+1}^{|L|} \vDash [\mathtt{x} = 2]$ *holds. We can combine this to derive* $\Gamma, L \vDash \mathbb{1}_{\mathtt{x}}; [\mathtt{x} = 2]$ *and consequently* $\Gamma \vDash t \ltimes \mathbb{1}_{\mathtt{x}}; [\mathtt{x} = 2]$*.*

One key observation is that each thread may observe the final write (i. e. with the greatest timestamp) to each shared variable. We can derive that every thread may perceive some nonempty list and the observations of all threads must hold for the list with the view mapping to the final writes. Thus, a disagreement is a contradiction, as stated in Theorem 3.13. We use this mostly to derive $false$ from $t \ltimes [false]$ for some thread t. Sometimes this also can be used to derive a contradiction when different threads assume different write orderings which disagree on some final write.

Theorem 3.13. *If there are a number of threads $t_i, \ldots, t_j$ with $t_i \ltimes I_i \wedge \cdots \wedge t_j \ltimes I_j$ and $I_i \wedge \cdots \wedge I_j \implies [false]$, then $(t_i \ltimes I_i \wedge \cdots \wedge t_j \ltimes I_j) \implies false$.*

Proof. Assume $t_i \ltimes I_i \wedge \cdots \wedge t_j \ltimes I_j$ and $I_i \wedge \cdots \wedge I_j \implies [false]$. For defining the final view, let $\delta_{max}(x) = w$ if and only if $var(w) = x$ and $w' \preceq w$ for every $w \in \sigma.writes$ with $var(w') = x$. By construction, $\delta \preceq \delta_{max}$ for every $\delta \in \sigma.Views$. Thus, $\langle \delta_{max} \rangle \in \sigma.Lists(\delta)$ for every $\delta \in \sigma.Views$. We can derive $\Gamma, \langle \delta_{max} \rangle \vDash I_i$ up to $\Gamma, \langle \delta_{max} \rangle \vDash I_j$. By definition of $\wedge$, we can derive $\Gamma, \langle \delta_{max} \rangle \vDash I_i \wedge \cdots \wedge I_j$. We know by our requirement that this implies $[false]$. Thus, $\Gamma, \langle \delta_{max} \rangle \vDash [false]$ holds. This is false; thus we can derive $(t_i \ltimes I_i \wedge \cdots \wedge t_j \ltimes I_j) \implies false$. □

3.5 Assertions on the Initial State $\Gamma_{\mathbf{Init}}$

In this section, we show which of these assertions hold on the initial state. The initial state $\Gamma_{\mathbf{Init}}$ is defined by the **Init** statements:

$$\mathbf{Init} = x_1 := k_1 \,;\, \cdots \,;\, x_n := k_n \,;\, [r_1 := l_1;] \ldots [r_m := l_m;]$$

The local variables and writes are defined to agree with these statements:

$$\begin{aligned} lst_{\mathbf{Init}}(t) : r_i \mapsto l_i & \quad \text{for each thread } t \text{ and its local variables } r_i \\ writes_{\mathbf{Init}} = \{(wr(x_i, k_i), 0) \mid 1 \leq i \leq n\} & \end{aligned}$$

To describe the interval in which specific writes occur, we introduce the following notation: $\mathbb{1}^x_{v_1 \ldots v_n} \equiv (\mathbb{1}_x \wedge [x = v_1]); \ldots ; (\mathbb{1}_x \wedge [x = v_n])$. Additionally we assume *atoms*(**Init**) to return the set of assignments and writes in **Init**. With this we define rules to derive statements holding in the initial state $\Gamma_{\mathbf{Init}}$.

$$\textsc{Init-asgn}\ \frac{r := v \in atoms(\mathbf{Init})}{\Gamma_{\mathbf{Init}} \vDash r = v} \qquad \textsc{Init-wr}\ \frac{x := v \in atoms(\mathbf{Init})}{\Gamma_{\mathbf{Init}} \vDash t \ltimes \mathbb{1}^x_v}$$

Theorem 3.14. *INIT-ASGN and INIT-WRare sound.*

Proof. Let **Init** $= x_1 := k_1 \,;\, \cdots \,;\, x_n := k_n \,;\, [r_1 := l_1;] \ldots [r_m := l_m;]$ and $\Gamma_{\mathbf{Init}} = (lst_{\mathbf{Init}}, \sigma_{\mathbf{Init}})$ be an according initial state.

INIT-ASGN holds by the definition of $lst_{\mathbf{Init}}$.

INIT-WR. Each view $\delta \in L$ in a list $L \in \sigma_{\mathbf{Init}}.Lists(tview_{\mathbf{Init}})$ must map shared variables to writes in $\sigma_{\mathbf{Init}}.writes$ of the same variable x_i. Thereby we know that every x_i is mapped to $\delta(x_i) = (wr(x_i, k_i), 0)$ by the definition of $writes_{\mathbf{Init}}$. Also for every shared variable there is only one write in $\sigma.writes$. With this, $\Gamma_{\mathbf{Init}}, L \vDash t \ltimes \mathbb{1}_{x_i}$ and $\Gamma_{\mathbf{Init}}, L \vDash t \ltimes [x = k_i]$ hold. We can derive $\Gamma_{\mathbf{Init}}, L \vDash t \ltimes \mathbb{1}^{x_i}_{k_i}$ □

4 Proof Rules

In this chapter, we present our proof rules and prove the soundness of some of them to give insights into the mechanics. The other soundness proofs work on the same principles, and are included in *Appendix C*. In Chapter 5, these rules are applied in examples, which do not require understanding of these inner workings.

We reason about validity of programs using Hoare triples, which we define below for partial correctness. This means we consider all possible final states of executions and ignore infinite executions. Proving that there are no infinite executions is considered in total correctness, which is not topic of this thesis.

Definition 4.1. *For assertions p and q, a parallel program P, and the empty program E, mapping each thread identifier to **skip**, we define the validity of a Hoare triple as follows:*

$$\begin{aligned} \vDash \{p\}\ P\ \{q\} \qquad & \textit{iff } \Gamma' \vDash q \textit{ for every } \Gamma' \in \Sigma_{C11} \textit{ with } (P, \Gamma) \Longrightarrow^{*} (E, \Gamma') \\ & \textit{for some } \Gamma \in \Sigma_{C11} \textit{ with } \Gamma \vDash p \end{aligned}$$

*For a parallel program with an **Init** statement we additionally require $\Gamma_{\textbf{\textit{Init}}} \vDash p$ to hold.*

To be able to refer to the executing thread in proof rules, we denote it in the index of the operation, e. g. $x :=_t e$ for a writing thread t.

4.1 Classical Proof Rules

Figure 4.1 shows classical proof rules for compound statements, logical reasoning and parallel composition. Classical proof rules for compound statements (combining multiple statements to non-atomic statements) hold by the same reasoning as for Hoare Logic [Hoa69, AdBO09]. The logical rules other than Disj2 are included in [AdBO09] and hold by reasoning over the transition system and are independent of what specific transition steps are possible. Disj2 can be easily derived by Disj1 and Cons.

Seq
$$\frac{\{p\}\ C_1\ \{r\} \qquad \{r\}\ C_2\ \{q\}}{\{p\}\ C_1\,;C_2\ \{q\}}$$

If
$$\frac{\{p \wedge b\}\ C_1\ \{q\} \qquad \{p \wedge \neg b\}\ C_2\ \{q\}}{\{p\}\ \textbf{if}\, b\, \textbf{then}\, C_1\, \textbf{else}\, C_2\, \textbf{fi}\ \{q\}}$$

While
$$\frac{\{p \wedge b\}\ C\ \{p\}}{\{p\}\ \textbf{while}\, b\, \textbf{do}\, C\, \textbf{od}\ \{p \wedge \neg b\}}$$

Until
$$\frac{\{p\}\ C\ \{r\} \qquad \{r\}\ \textbf{while}\, \neg b\, \textbf{do}\, C\, \textbf{od}\ \{r \wedge b\}}{\{p\}\ \textbf{do}\, C\, \textbf{until}\, b\, \textbf{od}\ \{r \wedge b\}}$$

Aux
$$\frac{\{p\}\ C\ \{q\} \qquad \mathit{vars}(q) \cap V = \emptyset}{\{p\}\ C_0\ \{q\}}$$

Parallel
$$\frac{\text{Proof outlines } \{p_i\}\ C_i\ \{q_i\} \text{ are interference free}}{\{\bigwedge_{i=1}^{n} p_i\}\ C_1 \| \ldots \| C_n\ \{\bigwedge_{i=1}^{n} q_i\}}$$

Cons
$$\frac{p \Rightarrow p' \qquad \{p'\}\ C\ \{q'\} \qquad q' \Rightarrow q}{\{p\}\ C\ \{q\}}$$

Conj
$$\frac{\{p_1\}\ C\ \{q_1\} \qquad \{p_2\}\ C\ \{q_2\}}{\{p_1 \wedge p_2\}\ C\ \{q_1 \wedge q_2\}}$$

Disj1
$$\frac{\{p_1\}\ C\ \{q\} \qquad \{p_2\}\ C\ \{q\}}{\{p_1 \vee p_2\}\ C\ \{q\}}$$

Disj2
$$\frac{\{p_1\}\ C\ \{q_1\} \qquad \{p_2\}\ C\ \{q_2\}}{\{p_1 \vee p_2\}\ C\ \{q_1 \vee q_2\}}$$

Figure 4.1: Classical proof rules

The Aux rule is the standard proof rule to remove auxiliary variables from a proof outline [AdBO09]. Here we obtain C_0 from C by removing all assignments to a set of auxiliary variables V. Statements of the form $\langle c, \mathit{aux} := e\rangle$ for an auxiliary variable $\mathit{aux} \in V$ for example are replaced by the statement c, removing the auxiliary assignment. Further, this is only allowed if no auxiliary variable $\mathit{aux} \in V$ occurs in C_0 any more, for example in another assignment or a condition.

The Parallel rule combines sequential programs into parallel programs the usual way [OG76, AdBO09]. Next, we define noninterference for this rule.

Definition 4.2. *A statement R with precondition $\mathit{pre}(R)$ (in the standard proof outline) does not interfere with an assertion p if $\{p \wedge \mathit{pre}(R)\}\ R\ \{p\}$.*

A *standard proof outline* of a sequential program is the program with exactly one assertion in $\{\}$ brackets before and after each statement. We call the assertion before a statement R its precondition $\mathit{pre}(R)$ and the assertion afterward its postcondition $\mathit{post}(R)$. Standard proof outlines are interference free if no statement of one proof outline interferes with any assertion of another.

A parallel proof outline is *valid* if the proof outlines of the sequential programs are valid, and the proof outlines are interference free. A proof outline of a sequential program is valid if for every statement R and C11 states $\Gamma, \Gamma' \in \Sigma_{C11}$ if $\Gamma \vDash \mathit{pre}(R)$ and $(R, \Gamma) \Longrightarrow^* (\textbf{skip}, \Gamma')$ implies $\Gamma' \vDash \mathit{post}(R)$.

4.2 Basic Rules

There are some basic rules shown below. They preserve assertions of which no free variable is changed in the statement. The set of free variables $fv(\varphi)$ for an assertion φ is the set of all local and shared variables used in the assertion and all shared variables x used in $\overline{A}(x)$, $\mathbb{1}_x$ or C_x in the assertion. These rules are (syntactically) equivalent to the rules Subst-asgn/Subst, Stable-ld/Stable-Ld, and Stable-wr/Stable-St in [LDW23]/[BDW24], where ld, wr, and st are shortcuts for load, write and store respectively. We later use them in proof outlines and to show that the rules ending on single are instances of their more complex versions, usually by preserving $false$.

$$\textsc{Subst-asgn}\quad \frac{}{\{\varphi(r := e)\}\ r :=_t e\ \{\varphi\}} \qquad \textsc{Stable-ld}\quad \frac{r \notin fv(\varphi)}{\{\varphi\}\ r \leftarrow_t x\ \{\varphi\}} \qquad \textsc{Stable-wr}\quad \frac{x \notin fv(\varphi)}{\{\varphi\}\ x :=_t e\ \{\varphi\}}$$

Theorem 4.3. *Subst-asgn, Stable-ld and Stable-wr are sound.*

This holds by induction over the structure of the formula, as shown in Appendix C.1.

4.3 Load Rules

In this section, we introduce our load rules. Write transition steps change the set of writes; therefore we handle loads first. Lemma 4.4 states that for every view, there is a specific part of an interval assertion that must hold for that view. We use this in the soundness proofs below to reason about interval assertions of the form $I_1; I_2$.

Lemma 4.4. *For a state $\Gamma = (lst, \sigma)$ with $\Gamma \vDash I_1; I_2$ and any views $\delta', \delta \in \sigma.Views$, the following holds:*

$$\Gamma, L_1 \cdot \langle\delta\rangle \vDash I_1 \textit{ for all lists of the form } L = L_1 \cdot \langle\delta\rangle \cdot L_2 \textit{ in } \sigma.Lists(\delta') \textit{ or}$$
$$\Gamma, \langle\delta\rangle \cdot L_2 \vDash I_2 \textit{ for all lists of the form } L = L_1 \cdot \langle\delta\rangle \cdot L_2 \textit{ in } \sigma.Lists(\delta')$$

Proof. By contradiction. Let $L = L_1 \cdot \langle\delta\rangle \cdot L_2 \in \sigma.Lists(\delta')$ such that $\Gamma, L_1 \cdot \langle\delta\rangle \nvDash I_1$. Let $L' = L_1' \cdot \langle\delta\rangle \cdot L_2' \in \sigma.Lists(\delta')$ such that $\Gamma, \langle\delta\rangle \cdot L_2' \nvDash I_2$. Then we can construct a list $L'' = L_1 \cdot \langle\delta\rangle \cdot L_2'$. Now $L'' \in \sigma.Lists(\delta')$ holds because both L and L' are also in $\sigma.Lists(\delta')$. L'' however can't fulfill $I_1; I_2$, because either the part before the split would have to begin with $L = L_1 \cdot \langle\delta\rangle$ or the part after the split has to end with $\langle\delta\rangle \cdot L_2$. By Lemma 3.7, that part cannot fulfill I_1 or I_2, respectively. □

Especially, an interval assertions of the form $I_1; I_2$ occurs in Ld-shift. The rules below are equivalent to the rules with the same name in [BDW24].

$$\textsc{Ld-shift}\quad \frac{\{t \ltimes I\}\ r \leftarrow_t x\ \{\psi\}}{\{t \ltimes [e(r := x)]; I\}\ r \leftarrow_t x\ \{e \vee \psi\}} \qquad \textsc{Ld-single}\quad \frac{}{\{t \ltimes [e(r := x)]\}\ r \leftarrow_t x\ \{e\}}$$

Theorem 4.5. *Ld-shift and Ld-single are sound.*

The intuition behind Ld-single is that an expression holding for x on all thread views possible after the transition step, after we read its value into r, the same expression must evaluate to *true* for r afterwards. For Ld-shift the new thread view after the transition step can be either in the first interval $[e(r := x)]$ or in the latter I. In the first case, similar to Ld-single, e must hold. Otherwise, we may assume that $t \ltimes I$ held before, because a transition from a state where it holds is possible.

Proof. Ld-shift
Let $\Gamma = (lst, \sigma)$ be the state before and $\Gamma' = (lst', \sigma')$ be the state after the transition step. There are only two changes by this transition: $\sigma.tview_t$ is updated to $\sigma'.tview_t$ and $lst(t)(r)$ is updated to $val(\sigma'.tview_t(x))$. All other thread views, all modification views, the covered set of the global state, σ and all other local variables of lst are the same in the state Γ before the read transition and Γ' after it.

With this we can continue to prove the soundness of Ld-shift. By Lemma 4.4 we know that one of the following holds for every $L = L_1 \cdot \langle \sigma'.tview_t \rangle \cdot L_2 \in \sigma.Lists(t)$:

(i) $(lst, \sigma), L_1 \cdot \langle \sigma'.tview_t \rangle \vDash t \ltimes [e(r := x)]$.
We can derive $[\![e(r := x)]\!]_{(lst,\sigma),\sigma'.tview_t} = true$. By the transition rules we know that $\sigma'.ls(t) = \sigma.ls(t)[r := tview_t(x)]$. By structural induction over the formula, we can conclude $[\![e]\!]_{lst'} = true$. For this let $\delta = \sigma'.tview_t$.

(a) $[\![\mathtt{a}(r := x)]\!]_{(lst,\sigma),\delta} = \begin{cases} val(\delta(x)) & \text{if } \mathtt{a} = r \\ lst(\mathtt{a}) & \text{otherwise} \end{cases} = lst'(\mathtt{a}) = [\![\mathtt{a}]\!]_{(lst',\sigma)}$

(b) $[\![\neg e(r := x)]\!]_{(lst,\sigma),\delta} = \neg\, [\![e(r := x)]\!]_{(lst,\sigma),\delta} = \neg\, [\![e]\!]_{(lst',\sigma)} = [\![\neg e]\!]_{(lst',\sigma)}$

(c) $[\![e_1(r := x) \oplus e_2(r := x)]\!]_{(lst,\sigma),\delta} = [\![e_1(r := x)]\!]_{(lst,\sigma),\delta} \oplus [\![e_2(r := x)]\!]_{(lst,\sigma),\delta} = [\![e_1]\!]_{(lst',\sigma)} \oplus [\![e_2]\!]_{(lst',\sigma)} = [\![e_1 \oplus e_2]\!]_{(lst',\sigma)}$
for $\oplus \in \{+, -, *, \div, \ldots, =, \leq, \geq, \ldots, \wedge, \vee, \ldots\}$

With this, $(lst', \sigma) \vDash e$ holds. Then $(lst', \sigma') \vDash e$ holds because σ and σ' are not used to evaluate e.

(ii) $(lst, \sigma), \langle \sigma'.tview_t \rangle \cdot L_2 \vDash t \ltimes I$.
By Lemma 3.7, $L_2 \vDash t \ltimes I$. For L_2 the same restrictions hold as for any list in $\sigma'.Lists(t)$. Therefore, the set of Lists allowed for L_2 is $\sigma'.Lists(t)$. With this we can derive $(lst, \sigma') \vDash t \ltimes I$. $(r \leftarrow_t x, lst, \sigma') \Longrightarrow (\mathbf{skip}, lst', \sigma')$ is possible by a transition, therefore $(lst', \sigma') \vDash \psi$ holds by the precondition.

We can conclude $(lst', \sigma') \vDash e \vee \psi$, which makes Ld-shift sound.

Ld-single
This is an instance of Ld-shift for $I \equiv [false]$ and $\psi \equiv false$. $t \ltimes [false] \Rightarrow false$ holds by Theorem 3.13 and $\{false\}\ r \leftarrow_t x\ \{false\}$ holds by Stable-ld. □

4.4 Write Rules

In this section we first introduce our general strategy to proof the soundness of our proof rules for writes. Then we show it on the example of Wr-top. Finally, we introduce our other write proof rules.

4.4.1 Soundness Proof Strategy

To show that a write proof rule is sound, we need to show that every state $\Gamma' = (lst', \sigma')$ after a write transition fulfills some assertion of the form $t \ltimes I'$. To do so, we take an arbitrary list L from $\sigma'.Lists(t)$ and show that it fulfills I'. For this, we replace the newly introduced write w with the write w_{prev} before it in L to produce a list $L[w/w_{prev}]$ which we know is in $\sigma.Lists(t)$, assuming $\Gamma = (lst, \sigma)$ is the state before the write transition. This is formalized in Lemma 4.7.

Example 4.6. *As an example we look at the following program, for executing* $\mathtt{x} := 1$ *after* $\mathtt{x} := 2$ *is already executed:*

$$\textbf{\textit{Init}} : \mathtt{x} := 0;$$

$$\begin{array}{l||l||l} \textbf{\textit{Thread } t}_{\textbf{\textit{1}}} & \textbf{\textit{Thread } t}_{\textbf{\textit{2}}} & \textbf{\textit{Thread } t}_{\textbf{\textit{3}}} \\ \mathtt{x} := 1 & \mathtt{x} := 2 & \textbf{\textit{skip}} \end{array}$$

Let the resulting state have the set $\sigma'.writes = \{w_0, w_1, w_2\}$ *for writes* $w_i = (wr(\mathtt{x}, i), i)$ *with value and timestamp* i*. In this state thread* t_3 *can observe for example the list* $L' = \langle \delta_0, \delta_1, \delta_2 \rangle \in \sigma'.Lists(t_3)$ *for views* $\delta_i : \mathtt{x} \mapsto w_i$*.*

Replacing the new write w_1 *by the previous one* w_0 *results in* $L = \langle \delta_0, \delta_0, \delta_2 \rangle$*. This list is obviously in* $\sigma.Lists(t_3)$*, because it does not contain the newly introduced write and all views succeed all previous ones.*

Finally, for the part around the replacement, the same interval assertions as before hold, e. g. $[\mathtt{x} = 0]$ *for* $\langle \delta_0 \rangle = L|_1^1 = L'|_1^1$ *and* $[\mathtt{x} = 2]$ *for* $\langle \delta_2 \rangle = L|_3^3 = L'|_3^3$*. For the part* $L'|_2^2 = \langle \delta_1 \rangle$ *containing the newly introduced write* w *holds* $\mathbb{1}_e^{\mathtt{x}}$*. We can combine these together into* $[\mathtt{x} = 0]; \mathbb{1}_1^{\mathtt{x}}; [\mathtt{x} = 2]$*, which holds for* L'*.*

Our general proof strategy shows that such a replacement works for all lists in $\sigma'.Lists(t)$*, allowing us to guarantee such a newly constructed interval assertion.*

First we need to formally define what it means to replace a write in a view δ and list $L = \langle \delta_1, \dots, \delta_{|L|} \rangle$. For a view δ, if it contains w, we replace that write with some other write, usually w_{prev}, which is the write directly before the newly introduced write w. Otherwise the view remains unchanged. We lift this to lists such that for each view in the list, the write w is replaced.

$$\delta[w/w_{prev}] := \begin{cases} \delta[var(w) := w_{prev}] & \text{if } \delta(var(w)) = w \\ \delta & \text{otherwise} \end{cases}$$

$$L[w/w_{prev}] := \langle \delta_1[w/w_{prev}], \dots, \delta_{|L|}[w/w_{prev}] \rangle$$

Let $\Gamma = (lst, \sigma)$ be the state before and $\Gamma' = (lst', \sigma')$ be the state after the write transition step. Let $w = (a, q)$ be the write written by the transition step and w_{prev} be the write previous to w, which is $w_{prev} = max_{q'}\{(a', q') \in \sigma.writes \mid var(a') = var(w) \wedge q' < tst(w)\}$. This write always exist because all newly introduced writes have a timestamp strictly greater than some other timestamp by $\sigma.fresh$. We use this replacement in Lemma 4.7 to create a list for which we know that it was in $\sigma.Lists(t)$. With this we know that it fulfills the precondition of the Hoare triple which we use in most proofs for write rules.

Lemma 4.7. *Let σ, σ' be arbitrary global states with $\sigma \overset{a}{\rightsquigarrow}_t \sigma'$ for $a \in \{wr(x, v_W), wr^R(x, v_W), upd^{RA}(x, v_R, v_W)\}$ and L' be an arbitrary list from $\sigma'.Lists(t)$. Then, for the write $w = \sigma'.tview_t(var(a))$ introduced in the transition and the write before it $w_{prev} = max_{q'}\{(a', q') \in \sigma.writes \mid var(a') = var(w) \wedge q' < tst(w)\}$, holds:*

$$L'[w/w_{prev}] \in \sigma.Lists(t)$$

Proof. Let $L = L'[w/w_{prev}]$. $L(i) \in \sigma.Views$ holds because all instances of the only new write in σ', w, are replaced by w_{prev} which is in $\sigma.writes$ by definition. With Lemma 3.9 we only need to derive $L(i) \preceq L(i+1)$ and $\sigma.tview_t \preceq L(i)$ to show $\delta \preceq \delta_i \preceq \delta_j$ for all $1 \leq i \leq j \leq n$, the other requirement for L to be in $\sigma.Lists(t)$.

First we need to show $L(i) \preceq L(i+1)$. This holds by the definition of $\sigma'.Lists$. There are four cases:

1. $L'(i)(x) = w$ and $L'(i+1)(x) = w$:
   $L(i) = L'(i)[x := w_{prev}] \preceq L'(i+1)[x := w_{prev}] = L(i+1)$
2. $L'(i)(x) \neq w$ and $L'(i+1)(x) = w$:
   $L(i) = L'(i) \preceq L'(i+1)[x := w_{prev}] = L(i+1)$ because $w' \preceq w_{prev}$ for all $w' \prec w$
3. $L'(i)(x) = w$ and $L'(i+1)(x) \neq w$:
   $L(i) = L(i)[x := w_{prev}] \preceq L'(i+1) = L(i+1)$ because $w_{prev} \preceq w$
4. $L'(i)(x) \neq w$ and $L'(i+1)(x) \neq w$:
   $L(i) = L'(i) \preceq L'(i+1) = L(i+1)$

We conclude $L(i) \preceq L(i+1)$ in every case.

Finally we need to show $\sigma.tview_t \preceq L(i)$. For $L(i) \in \sigma.Views$, there are two cases:

1. $L'(i)(x) \neq w$: $\sigma.tview_t \preceq \sigma'.tview_t$ by Theorem 3.1 and $\sigma'.tview_t \preceq L'(i)$ by definition of $\sigma'.Lists$. $L'(i)[w/w_{prev}] = L'(i) = L(i)$ because $L'(i)(x) \neq w$. We can conclude $\sigma.tview_t \preceq L(i)$.
2. $L'(i)(x) = w$: w_{prev} is the write selected from $\sigma.OW(t, x)$, since $\sigma.fresh$ guarantees that the new write w has the next timestamp. $\sigma.tview_t \preceq \sigma'.tview_t\,[x := w_{prev}]$ holds since $w_{prev} \in \sigma.OW(t, x)$. Further $\sigma'.tview_t\,[x := w_{prev}] \preceq L'(i)\,[x := w_{prev}]$

holds because $\sigma'.\mathit{tview}_t \preceq L'(i)$ since $L' \in \sigma'.\mathit{Lists}(t)$. Also $L'(i)\,[x := w_{prev}] = L(i)$ by definition of L. We can derive $\sigma.\mathit{tview}_t \preceq L(i)$.

We conclude $\sigma.\mathit{tview}_t \preceq L(i)$ in either case.

By Lemma 3.9 this is sufficient to prove $\delta \preceq \delta_i \preceq \delta_j$ for all $1 \leq i \leq j \leq n$. With $L(i) \in \sigma.\mathit{Views}$, we can derive $L \in \sigma.\mathit{Lists}(t)$. □

Now, we know $L'[w/w_{prev}] \in \sigma.\mathit{Lists}(t)$. We use this in our soundness proofs to split the list in parts with and without the new write w. $\mathbb{1}^x_e$ holds trivially for the parts with the new write w, because it is the only write to x in that part and its value is e. For the parts without the new write, interval assertions fulfilled by Γ, L are also fulfilled by Γ', L. This is formalized in Lemma 4.8.

Lemma 4.8. *For* Write *and* Update *transition steps with the C11 states Γ before and Γ' after the transition step, if $L \in \sigma.\mathit{Lists}(t')$ for some thread t' and I is an interval formula, then $\Gamma, L \vDash I$ implies $\Gamma', L \vDash I$.*

The proof by induction over the formula is included in Appendix B.4. The intuition is that for each view, extended expressions are evaluated the same, and for each list, if an interval assertion held before the transition step, it still holds afterwards.

This is our general strategy to prove the soundness of our proof rules. Because we cannot derive useful postconditions with this alone, we need multiple proof rules with different, stronger preconditions. In the next section, we show the soundness of Wr-top as an example, where we restrict the writing thread t to only observes a single write on the written variable x in the precondition: $t \ltimes \mathbb{1}_x$.

4.4.2 Wr-top

Below we introduce Wr-top, a proof rule requiring the writing thread to perceive only a single write. This requires the thread view of that thread to map to the newest write to that variable and create a new write with an even greater timestamp. Thus, in the postcondition of the Hoare triple, we may insert the new write at the end. To describe this write, we use the previously established notation $\mathbb{1}^x_e \equiv \mathbb{1}_x \wedge [x = e]$. This proof rule is similar to Wr-other-1 from [LDW23], which has an additional assertion $R(x)$ for older writes, and St-Other1 from [BDW24], which specifies the writing thread with $x.\texttt{tid} = t$. On the other hand, we guarantee that there is only one new write with $\mathbb{1}_x$.

Wr-top

$$\frac{}{\{t \ltimes \mathbb{1}_x \wedge t' \ltimes I\}\ x :=_t e\ \{t' \ltimes I; \mathbb{1}^x_e\}}$$

Theorem 4.9. *Wr-top is sound.*

Proof. Let $w_{max} := max_q\{(a,q) \in \sigma.writes \mid var(a) = x\}$ be the write to x with the maximal timestamp. Then $\langle \sigma.tview_t, \sigma.tview_t[x := w_{max}]\rangle \in \sigma.Lists(t)$ because $\sigma.tview_t(x) \preceq w_{max}$. With $t \ltimes \mathbb{1}_x$ we can derive $\sigma.tview_t(x) = w_{max}$.

Write guarantees with $\sigma.fresh$ that the newly introduced write w has an even greater timestamp than the previously latest write: $tst(w) > tst(\sigma.tview_t(x))$. Because it is the only new write ($\sigma'.writes = \sigma.writes \cup \{w\}$), it succeeds all other writes to x in σ: $tst(w') < tst(w)$ for all $w' \in \sigma'.writes$ with $var(w') = x$. Therefore, no other write to x can occur in L' after w in any $L' \in \sigma'.Lists(t)$.

Let $0 \leq k \leq |L'$ such that $w \notin L'|_1^k.writes$ and $\delta(x) = w$ for every $\delta \in L'|_{k+1}^{|L'|}$. We know that $L'[w/w_{prev}] \in \sigma.Lists(t)$ by Lemma 4.7, therefore also $L'|_1^k[w/w_{prev}] \in \sigma.Lists(t)$. Further, by definition of k, $L'|_1^k[w/w_{prev}] = L'|_1^k$ because $w \notin L'|_1^k.writes$. Therefore, we can derive $\Gamma, L'|_1^k \vDash I$ and by Lemma 4.8 $\Gamma', L'|_1^k \vDash I$.

$\Gamma', L'|_{k+1}^{|L'|} \vDash \mathbb{1}_x \wedge [x = e]$ holds trivially because $\delta(x) = w$ for every $\delta \in L'|_{k+1}^{|L'|}$. We can derive $\Gamma', L'|_1^k \cdot L'|_{k+1}^{|L'|} \vDash I; \mathbb{1}_e^x$. As $L'|_1^k \cdot L'|_{k+1}^{|L'|} = L'$, this is what we needed to show: $\Gamma', L' \vDash I; \mathbb{1}_e^x$. □

4.4.3 Additional Write Rules

In combination with Wr-top from the previous chapter, we often use Wr-own-1wr to preserve that the writing thread perceives only one write (to the written variable). The rules Wr-Own from [LDW23] and Un from [BDW24] are similar. The former holds for a stronger memory model where the precondition *true* is sufficient to derive the postcondition $t \ltimes [x = e]$. The latter is syntactically equivalent to our rule if we restrict our rule to $I \equiv [false]$, thereby removing $;I$, and ignore the additional guarantees $x.\texttt{tid} = t$ they provide, assuming $t \ltimes \mathbb{1}_x$ and $t \uparrow x$ are equivalent.

Additionally, in cases with more than one writing thread the rule Wr-1wr can be used, of which Wr-1wr-single is a simplified and weaker version.

Wr-own-1wr

$$\frac{}{\{t \ltimes \mathbb{1}_x; I\}\ x :=_t e\ \{t \ltimes \mathbb{1}_e^x; I\}}$$

Wr-1wr-single

$$\frac{}{\{t' \ltimes \mathbb{1}_x \wedge I\}\ x :=_t e\ \{t' \ltimes I; \mathbb{1}_e^x\}}$$

Wr-1wr

$$\frac{\{t' \ltimes I_2\}\ x :=_t e\ \{t' \ltimes I_2'\}}{\{t' \ltimes (I_1 \wedge \mathbb{1}_x); I_2\}\ x :=_t e\ \{t' \ltimes ((I_1; \mathbb{1}_e^x; I_2) \vee (I_1; I_2'))\}}$$

Theorem 4.10. *Wr-own-1wr, Wr-1wr and Wr-1wr-single are sound.*

The soundness proofs are similar to that for Wr-top and are given in Appendix C.2. For Wr-own-1wr we use that the current thread view is updated to the new write, thus replacing the write which can only occur at the beginning of the list, $\sigma.tview_t(x)$. For Wr-1wr we use that the new write w can only be inserted at one point in a list, without any other writes in between views with w. Wr-1wr-single is an instance of Wr-1wr, similar to how Ld-single is an instance of Ld-shift.

4.5 Release/ Acquire

For release acquire we need additional rules for the new operations. Because all assumptions made proving the previous rules also hold for acquiring reads and releasing writes, they can be applied the same way to those operations. Additionally, the new rules below can be used, which are not sound for the relaxed reads and writes. Here $\overline{A}(x)$ describes a view which cannot occur after an acquiring read from x, as stated in Lemma 3.4 and argued for in Section 3.2. We introduce $\overline{A}^x_e$ as a shortcut for $\overline{A}^x_e \equiv [\overline{A}(x)] \wedge \mathbb{1}^x_e$, denoting that there is only one write and it cannot occur after an acquiring read. This holds when the reading thread observes a write, but its thread view does not succeed the modification view of that write.

Ld-a-shift

$$\frac{\{t \ltimes I\}\ r \leftarrow^A_t x\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ r \leftarrow^A_t x\ \{\psi\}}$$

Wr-r-top

$$\frac{\{t \ltimes I_t\}\ x :=^R_t e\ \{t \ltimes I'_t\}}{\{t \ltimes (I_t \wedge \mathbb{1}_x) \wedge t' \ltimes I\}\ x :=^R_t e\ \{t' \ltimes I; \overline{A}^x_e; I'_t\}}$$

Theorem 4.11. *Ld-a-shift and Wr-r-top are sound.*

The soundness proofs are given in Appendix C.3. Ld-a-shift details that $\overline{A}(x)$ can be ignored at the beginning of an assertion if the perceiving thread reads that variable. The intuition is that because this cannot hold on the thread view after an acquiring read, we know that we can add this thread view at the beginning of each list. Then $\overline{A}(x)$ can only hold on the empty list and the rest of the interval assertion must hold on the full list. When Wr-r-top can be applied, the new write can only occur at the end of its interval, because the writing thread observes only one write, similar to Wr-top. Then, after the thread view of the reading thread succeeds the modification view (i. e. $\overline{A}(x)$ does not hold any more), its lists are a subset of the writing threads and therefore, the writing threads interval assertion must hold.

4.6 Swap

All rules for relaxed and releasing writes may also be used for swap operations, because all assumptions made during the proofs also are valid for it. Additionally, we need proof rules both for the swap operation and for covered writes it introduces. For the latter, we use the interval assertion C_x, asserting that all writes to x in a list are in the set $\sigma.covered$ of the global state σ. These rules have a small similarity with Swap-skip from [LDW23], in that both rules allow ignoring a leading interval assertion.

Swap-a-shift

$$\frac{\{t \ltimes I\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{\psi\}}$$

Wr-cvd

$$\frac{\{t \ltimes I\}\ x :=_t e\ \{\psi\}}{\{t \ltimes C_x; I\}\ x :=_t e\ \{\psi\}}$$

Theorem 4.12. *Swap-a-shift and Wr-cvd are sound.*

The soundness proofs are given in Appendix C.4. Swap-a-shift is identical to Ld-a-shift, except that it can be applied to swaps instead of reads. Wr-cvd describes that no write can be inserted directly after a write which is already covered by an atomic swap operation.

Additionally, we need rules allowing us to derive C_x when a swap operation is executed. The two rules below guarantee the written value and the passing of an interval assertion respectively.

Swap-wr

$$\frac{}{\{t \ltimes C_x; \mathbb{1}_x \land t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{t' \ltimes C_x; \mathbb{1}_v^x\}}$$

Swap-r

$$\frac{\{t \ltimes I_t\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{t \ltimes I'_t\}}{\{t \ltimes (I_t \land C_x; \mathbb{1}_x) \land t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{t' \ltimes C_x; \overline{A}_v^x; I'_t\}}$$

Theorem 4.13. *Swap-wr and Swap-r are sound.*

The soundness proofs are given in Appendix C.5. These are the proof rules for writing a value with a swap. The precondition guarantees that both the writing thread t and perceiving thread t' observe only a single uncovered write to x, which allows us to derive that this write is added to the *covered* set and the only write they might perceive afterwards is the new one, since it is the only one not in *covered*. Other than that, Swap-wr is very similar to Wr-top and Swap-r to Wr-r-top.

5 Examples

After introducing our proof rules in the previous chapter, we show how they can be used in validity proofs for proof outlines. Classical proof rules are explained in Section 4.1, and we recall our new rules before they are first used. There is also a quick reference in Appendix A, where all proof rules are printed. We showcase most proof rules only once, for full validity proofs see Appendix D. We begin with several standard litmus tests, which are minimal examples showing of behavior of different memory models. In the end, we also verify Peterson's algorithm as a case study.

5.1 Load Buffering

In this section, we take a look at Load Buffering, a simple litmus test to showcase some of the simpler rules. Our new rules we use here are the following:

$$\textsc{Stable-ld}\ \frac{r \notin fv(\varphi)}{\{\varphi\}\ r \leftarrow_t x\ \{\varphi\}} \qquad \textsc{Stable-wr}\ \frac{x \notin fv(\varphi)}{\{\varphi\}\ x :=_t e\ \{\varphi\}} \qquad \textsc{Ld-single}\ \frac{}{\{t \ltimes [e(r := x)]\}\ r \leftarrow_t x\ \{e\}}$$

Theorem 5.1. *The proof outline for Load Buffering in Figure 5.1 is valid.*

$$\begin{array}{c}
\textbf{Init} : \mathtt{x} := 0\,;\mathtt{x} := 0\,;\mathtt{a} := 0\,;\mathtt{b} := 0;\\
\{t_1 \ltimes [\mathtt{x} = 0] \land t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{a} = 0 \land \mathtt{b} = 0\}\\
\begin{array}{l||l}
\textbf{Thread } \mathbf{t_1} & \textbf{Thread } \mathbf{t_2}\\
\{t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{b} = 0\} & \{t_1 \ltimes [\mathtt{x} = 0] \land \mathtt{a} = 0\}\\
1 : \mathtt{a} \leftarrow \mathtt{x}; & 3 : \mathtt{b} \leftarrow \mathtt{y};\\
\{t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{b} = 0\} & \{t_1 \ltimes [\mathtt{x} = 0] \land \mathtt{a} = 0\}\\
2 : \mathtt{y} := 1 & 4 : \mathtt{x} := 1\\
\{\mathtt{a} = 0 \lor \mathtt{b} = 0\} & \{\mathtt{a} = 0 \lor \mathtt{b} = 0\}
\end{array}\\
\{\mathtt{a} = 0 \lor \mathtt{b} = 0\}
\end{array}$$

Figure 5.1: Proof outline for Load Buffering, with $\mathtt{a}, \mathtt{b} \in \mathit{Var}_L$ and $\mathtt{x}, \mathtt{y} \in \mathit{Var}_G$

The full proof can be seen in Appendix D.1, but here we showcase where the new proof rules are used. The proof outline is very similar to the proof outline in [DDDW20], and holds the same way in Piccolo for other (weak) memory models with these proof rules, such as SC, TSO, and SRA [LDW23, BDW24].

We use STABLE-LD and STABLE-WR to derive the validity of the sequential programs of the threads. For example, below STABLE-LD preserves an assertion only containing the free variables `y` and `b` for loading a value into `a`. Writing to `y` on the other hand only preserves $\mathtt{b} = 0$, but this is sufficient to derive the postcondition $\mathtt{a} = 0 \vee \mathtt{b} = 0$.

1 by STABLE-LD:

$\{t_2 \ltimes [\mathtt{y} = 0] \wedge \mathtt{b} = 0\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{t_2 \ltimes [\mathtt{y} = 0] \wedge \mathtt{b} = 0\}$

2 by STABLE-WR, CONS:

$\{t_2 \ltimes [\mathtt{y} = 0] \wedge \mathtt{b} = 0\}$
$\{\mathtt{b} = 0\}$
$\mathtt{y} := 1$
$\{\mathtt{b} = 0\}$
$\{\mathtt{a} = 0 \vee \mathtt{b} = 0\}$

We need LD-SINGLE only to show noninterference. This means that, even if a statement of a thread is executed, the assertions of the other threads are preserved. Below we show that the assertion $t_2 \ltimes [\mathtt{x} = 0] \wedge \mathtt{b} = 0$ of t_2 is preserved when statement 1 (of t_1) is executed. Here $t_1 \ltimes [\mathtt{x} = 0]$ can be preserved by STABLE-LD. On the other hand, we can derive $\mathtt{b} = 0$ afterwards by LD-SINGLE only because we also know that $t_1 \ltimes [\mathtt{x} = 0]$ holds before the execution.

(I) by STABLE-LD:

$\{t_1 \ltimes [\mathtt{x} = 0]\}\mathtt{a} \leftarrow \mathtt{x}\{t_1 \ltimes [\mathtt{x} = 0]\}$

(II) by LD-SINGLE:

$\{t_1 \ltimes [\mathtt{x} = 0]\}\mathtt{a} \leftarrow \mathtt{x}\{\mathtt{a} = 0\}$

by (I), (II), CONJ and CONS:

$\{t_2 \ltimes [\mathtt{y} = 0] \wedge \mathtt{b} = 0 \wedge t_1 \ltimes [\mathtt{x} = 0] \wedge \mathtt{a} = 0\}$
$\{t_1 \ltimes [\mathtt{x} = 0] \wedge t_1 \ltimes [\mathtt{x} = 0]\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{t_1 \ltimes [\mathtt{x} = 0] \wedge \mathtt{a} = 0\}$

5.2 RRC2

In this section, we take a look at RRC2 (read-read coherence), which reasons about coherence of writes of one thread to the one variable. For this we use the rules below in addition to rules shown in the previous section.

WR-TOP

$$\frac{}{\{t \ltimes \mathbb{1}_x \wedge t' \ltimes I\}\ x :=_t e\ \{t' \ltimes I; \mathbb{1}_e^x\}}$$

WR-OWN-1WR

$$\frac{}{\{t \ltimes \mathbb{1}_x; I\}\ x :=_t e\ \{t \ltimes \mathbb{1}_e^x; I\}}$$

LD-SHIFT

$$\frac{\{t \ltimes I\}\ r \leftarrow_t x\ \{\psi\}}{\{t \ltimes [e(r := x)]; I\}\ r \leftarrow_t x\ \{e \vee \psi\}}$$

$$\textbf{Init} : \mathtt{x} := 0;$$
$$\{t_1 \ltimes \mathbb{1}_{\mathtt{x}} \wedge t_2 \ltimes [\mathtt{x} = 0]\}$$

$$\begin{array}{l||l}
\textbf{Thread } \mathbf{t_1} & \textbf{Thread } \mathbf{t_2} \\
\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\} & \{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 2]\} \\
1 : \mathtt{x} := 1; & 3 : \mathtt{a} \leftarrow \mathtt{x}; \\
\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\} & \{\mathtt{a} = 2 \Rightarrow t_1 \ltimes [\mathtt{x} = 2]\} \\
2 : \mathtt{x} := 2 & 4 : \mathtt{b} \leftarrow \mathtt{x} \\
\{\mathit{true}\} & \{\mathtt{a} = 2 \Rightarrow \mathtt{b} = 2\}
\end{array}$$

$$\{\mathtt{a} = 2 \Rightarrow \mathtt{b} = 2\}$$

Figure 5.2: Proof outline of RRC2, with $\mathtt{a}, \mathtt{b} \in \mathit{Var}_L$ and $\mathtt{x} \in \mathit{Var}_G$

In Figure 5.2 we can see a proof outline for RRC2. We expect the initial value 0 for $\mathtt{x}$ and then writes with values 1 and 2. This could be described by $t_2 \ltimes [\mathtt{x} = 0]; [\mathtt{x} = 1]; [\mathtt{x} = 2]$, but here $t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 2]$ is sufficient. It also shortens the validity proof, because some proof rules must be applied iteratively on each interval. The assertion $t_1 \ltimes \mathbb{1}_{\mathtt{x}}$ implies that t_1 only perceives a single write. Because all threads may perceive the write with the maximal timestamp, t_1 must perceive only that write. Consequently, all new writes t_1 introduces must succeed that write and have a new maximal timestamp at their time of creation. This is what allows Wr-top to add new writes at the end of the interval some thread might perceive.

Theorem 5.2. *The proof outline for RRC2 in Figure 5.2 is valid.*

The full proof can be seen in Appendix D.2, here we showcase our new proof rules. With similar rules and when replacing $t_1 \ltimes \mathbb{1}_{\mathtt{x}}$ with $t_1 \uparrow \mathtt{x}$, the proof outline holds in Piccolo for SC, TSO, and SRA [LDW23, BDW24].

The same litmus test is also verified in [DDDW20], but their proof outline seems more complex. Instead of constructing a sequential proof for t_2, which guarantees $\mathtt{a} = 2 \Rightarrow \mathtt{b} = 2$ as postcondition, they derive as postconditions of t_1 and t_2 that all writes to $\mathtt{x}$ with value 2 succeed those with value 1 and $\mathtt{a} \neq \mathtt{b}$ implies that a write with value a precedes one with value b. Their combination allows them to derive the weaker postcondition $\mathtt{a} = 2 \implies \mathtt{b} = 2$ for the parallel program.

For our proof, if we look at the proof for statement 1 below, $\mathtt{x} \neq 2$ is preserved by Wr-top, because we additionally know $t_1 \ltimes \mathbb{1}_{\mathtt{x}}$ and $1 \neq 2$, which allows the intervals to be combined.

Wr-own-1wr allows us to derive that t_1 only perceives the newest write to $\mathtt{x}$ both before and after the transition. The intuition behind the rule is that the single write at the beginning of the interval before the write must observe the write $\sigma.\mathit{tview}_t(\mathtt{x})$ of the global state σ before the transition. Because the thread view of t is updated by the transition to observe the newly introduced write, we know that the old write cannot be observed anymore and that the new write must be at the beginning of the interval.

Note that $[\mathit{false}]$ is only fulfilled by the empty list and therefore can be freely added and removed when using it with the chop (;) operator.

(I) by Wr-top and Cons:

$\{t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$
$\mathtt{x} := 1$
$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 1]\}$
$\{t_2 \ltimes [\mathtt{x} \neq 2]\}$

(II) by Wr-own-1wr and Cons:

$\{t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$
$\{t_1 \ltimes \mathbb{1}_{\mathtt{x}}; [false]\}$
$\mathtt{x} := 1$
$\{t_1 \ltimes \mathbb{1}^{\mathtt{x}}_{1}; [false]\}$
$\{t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$

By (I), (II), Conj and Cons:

$$\{t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}\mathtt{x} := 1\{t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$$

The other newly introduced rule is Ld-shift. The intuition behind the rule is that a thread either reads from the current interval, or if it reads from a later interval, it won't be able to perceive the previous interval any more.

by Stable-ld, Ld-shift and Cons

$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 2]\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{a \neq 2 \lor t_2 \ltimes [\mathtt{x} = 2]\}$
$\{a = 2 \Rightarrow t_2 \ltimes [\mathtt{x} = 2]\}$

5.3 RRC

In this section, we take a look at RRC, a litmus test, which reasons about coherence of writes of different threads to the same variable. Because of this the rule Wr-1wr is necessary. Wr-1wr-single is a weaker version of this rule which allows us to prove the validity of proof outlines with fewer applications of proof rules.

Wr-1wr-single

$$\frac{}{\{t' \ltimes \mathbb{1}_x \land I\}\ x :=_t e\ \{t' \ltimes I; \mathbb{1}^x_e\}}$$

Wr-1wr

$$\frac{\{t' \ltimes I_2\}\ x :=_t e\ \{t' \ltimes I'_2\}}{\{t' \ltimes (I_1 \land \mathbb{1}_x); I_2\}\ x :=_t e\ \{t' \ltimes ((I_1; \mathbb{1}^x_e; I_2) \lor (I_1; I'_2))\}}$$

Theorem 5.3. *The proof outline for RRC in Figure 5.3 is valid.*

The full proof can be seen in Appendix D.3, but here we showcase where the new proof rules are used. The new rules are applied to guarantee that statement 1 does not interfere with the precondition of statement 3.

This proof outline is not applicable to Piccolo with other weak memory models, because there are no similar proof rules to Wr-1wr and Wr-1wr-single for them

$$\begin{array}{c}
\textbf{Init} : \mathtt{x} := 0; \\
\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{0}\}
\end{array}$$

$$\begin{array}{l||l||l||l}
\textbf{Thread } \mathbf{t_1} & \textbf{Thread } \mathbf{t_2} & \textbf{Thread } \mathbf{t_3} & \textbf{Thread } \mathbf{t_4} \\
\left\{\begin{array}{l} \mathtt{a} \neq 1 \wedge \\ t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02} \end{array}\right\} & \left\{\begin{array}{l} \mathtt{c} \neq 2 \wedge \\ t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{01} \end{array}\right\} & \{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \vee \mathbb{1}^{\mathtt{x}}_{021}\} & \{t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \vee \mathbb{1}^{\mathtt{x}}_{021}\} \\
1 : \mathtt{x} := 1 & 2 : \mathtt{x} := 2 & 3 : \mathtt{a} \leftarrow \mathtt{x}; & 5 : \mathtt{c} \leftarrow \mathtt{x}; \\
\{\mathit{true}\} & \{\mathit{true}\} & \{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\} & \{\mathtt{c} = 2 \Rightarrow t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{21}\} \\
 & & 4 : \mathtt{b} \leftarrow \mathtt{x} & 6 : \mathtt{d} \leftarrow \mathtt{x} \\
 & & \left\{\begin{array}{l} (a = 1 \wedge b = 2) \\ \quad \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2} \end{array}\right\} & \left\{\begin{array}{l} (c = 2 \wedge d = 1) \\ \quad \Rightarrow t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{1} \end{array}\right\}
\end{array}$$

$$\{(\mathtt{a} = 1 \wedge \mathtt{b} = 2 \wedge \mathtt{c} = 2) \Rightarrow \mathtt{d} \neq 1\}$$

Figure 5.3: Proof outline for RRC, with $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \in \mathit{Var}_L$ and $\mathtt{x} \in \mathit{Var}_G$

[DDDW22], but there are similar proof outlines with Piccolo for SRA [LDW23] and for C11 RAR with Owicki-Gries reasoning [DDDW20]. That SRA proof outline is very similar to ours, except swapping the values of the local variables and not needing the restrictions to one write ($\mathbb{1}_{\mathtt{x}}$), because of the stronger memory model. That C11 proof outline is different, and the differences are similar to those found with RRC2: They derive that writes are ordered one way in t_3 and another way in t_4. This, combined with the restriction that there are only one write with the values 1 and 2 each, derived in t_1 and t_2, leads to a $\mathtt{a} = 1 \wedge \mathtt{b} = \mathtt{c} = 1 \Rightarrow \mathtt{d} \neq 1$.

Below we begin with reasoning on the rightmost part of the interval, $t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{2}$. Here, by Wr-1wr-single, we can derive $t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{21}$. We can use this with Wr-1wr to guarantee that t_0 either perceives $\mathbb{1}^{\mathtt{x}}_{012}$ (resulting from $(I_1; \mathbb{1}^{\mathtt{x}}_{e}; I_2)$) or $\mathbb{1}^{\mathtt{x}}_{021}$ (resulting from $(I_1; I_2')$).

(I) by Wr-1wr-single and Cons

$$\begin{array}{l}
\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\} \\
\{t_0 \ltimes \mathbb{1}_{\mathtt{x}} \wedge [x = 2]\} \\
\mathtt{x} := 1 \\
\{t_0 \ltimes (\mathbb{1}_{\mathtt{x}} \wedge [x = 2]); (\mathbb{1}_{\mathtt{x}} \wedge [x = 1])\} \\
\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{21}\}
\end{array}$$

pre(3) by (I), Wr-1wr and Cons

$$\begin{array}{l}
\{\mathtt{a} \neq 1 \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02} \wedge t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \vee \mathbb{1}^{\mathtt{x}}_{021}\} \\
\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\} \\
\{t_0 \ltimes (\mathbb{1}_{\mathtt{x}} \wedge [x = 0]); \mathbb{1}^{\mathtt{x}}_{2}\} \\
\mathtt{x} := 1 \\
\left\{ t_0 \ltimes \left(\begin{array}{c} ((\mathbb{1}_{\mathtt{x}} \wedge [x = 0]); \mathbb{1}^{\mathtt{x}}_{1}; \mathbb{1}^{\mathtt{x}}_{2}) \vee \\ ((\mathbb{1}_{\mathtt{x}} \wedge [x = 0]); \mathbb{1}^{\mathtt{x}}_{21}) \end{array} \right) \right\} \\
\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \vee \mathbb{1}^{\mathtt{x}}_{021}\}
\end{array}$$

Further $t_3 \ltimes [\mathtt{x} = 2] \wedge t_4 \ltimes [\mathtt{x} = 1]$ is a contradiction by Theorem 3.13. With this the postcondition of the parallel program can be derived as seen below.

$$\begin{array}{l}
\quad \mathit{true} \wedge \mathit{true} \wedge ((\mathtt{a} = 1 \wedge \mathtt{b} = 2) \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}) \wedge ((\mathtt{c} = 2 \wedge \mathtt{d} = 1) \Rightarrow t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{1}) \\
\Longrightarrow (\mathtt{a} = 1 \wedge \mathtt{b} = 2 \wedge \mathtt{c} = 2 \wedge \mathtt{d} = 1) \Rightarrow (t_3 \ltimes [\mathtt{x} = 2] \wedge t_4 \ltimes [\mathtt{x} = 1]) \\
\Longrightarrow (\mathtt{a} = 1 \wedge \mathtt{b} = 2 \wedge \mathtt{c} = 2 \wedge \mathtt{d} = 1) \Rightarrow \mathit{false} \\
\Longrightarrow (\mathtt{a} = 1 \wedge \mathtt{b} = 2 \wedge \mathtt{c} = 2) \Rightarrow \mathtt{d} \neq 1
\end{array}$$

Init : $\mathtt{x} := 0\,;\, \mathtt{y} := 0;$

$\{t_0 \ltimes (\mathbb{1}_0^{\mathtt{x}} \wedge \mathbb{1}_0^{\mathtt{y}})\}$

$$
\begin{array}{l||l}
\textbf{Thread } \mathbf{t_1} & \textbf{Thread } \mathbf{t_2} \\
\{t_0 \ltimes (\mathbb{1}_0^{\mathtt{x}} \wedge \mathbb{1}_0^{\mathtt{y}})\} & \{t_2 \ltimes \mathbb{1}_0^{\mathtt{y}}; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\} \\
1 : \mathtt{x} := 1; & 3 : \mathtt{a} \leftarrow^{A} \mathtt{y}; \\
\{t_1 \ltimes ([\mathtt{x} = 1] \wedge \mathbb{1}_{\mathtt{y}}) \wedge t_2 \ltimes \mathbb{1}_0^{\mathtt{y}}\} & \{\mathtt{a} = 1 \Rightarrow t_2 \ltimes [\mathtt{x} = 1]\} \\
2 : \mathtt{y} :=^{R} 1 & 4 : \mathtt{b} \leftarrow \mathtt{x} \\
\{\mathit{true}\} & \{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}
\end{array}
$$

$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$

Figure 5.4: Proof outline for message passing, with $\mathtt{a}, \mathtt{b} \in \mathit{Var}_L$ and $\mathtt{x}, \mathtt{y} \in \mathit{Var}_G$

The reason is that there always exists a last view which is only succeeded by itself in $\sigma.\mathit{Views}$ for any global state σ. Both assertions must hold for this view. Thus, if there are multiple threads $t_i, ..., t_j$ with $t_i \ltimes I_i \wedge \cdots \wedge t_j \ltimes I_j$ and $t' \ltimes I_i \wedge \cdots \wedge I_j \implies [\mathit{false}]$, then also $t_i \ltimes I_i \wedge \cdots \wedge t_j \ltimes I_j$ is a contradiction.

5.4 Message Passing

In this section, we take a look at Message Passing, a litmus test with a releasing write and an acquiring read. In addition to the rules for relaxed reads and writes, the rules below can be used for releasing writes and acquiring reads.

$\overline{A}(x)$ describes a view δ that cannot occur directly after an acquiring read, because some write $\delta(y)$ has a smaller timestamp than the write $\mathit{mview}_{\delta(x)}(y)$ to the same variable in the corresponding modification view, to which that acquiring read would update it. When we can guarantee $\overline{A}(x)$, it usually holds for exactly one write, which we know the value of. Therefore, we use the shortcut $\overline{A}_e^{\mathtt{x}} \equiv [\overline{A}(\mathtt{x})] \wedge \mathbb{1}_e^{\mathtt{x}}$.

$$
\textsc{Wr-R-top}\quad \frac{\{t \ltimes I_t\}\ x :=_t^R e\ \{t \ltimes I_t'\}}{\{t \ltimes (I_t \wedge \mathbb{1}_x) \wedge t' \ltimes I\}\ x :=_t^R e\ \{t' \ltimes I; \overline{A}_e^x; I_t'\}}
\qquad
\textsc{Ld-A-shift}\quad \frac{\{t \ltimes I\}\ r \leftarrow_t^A x\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ r \leftarrow_t^A x\ \{\psi\}}
$$

Theorem 5.4. *The proof outline for message passing in Figure 5.4 is valid.*

The full proof can be seen in Appendix D.4, but here we showcase where the new proof rules are used. There are a different proof outline for message passing in C11 [DDDW20], and slightly different ones in Piccolo for other weak memory models [LDW23, BDW24]. In Piccolo, because there are no different types of load operations in the used memory models, they do not require the synchronization part $\overline{A}(\mathtt{y})$, which tells us that we need an acquiring read to finish the synchronization in C11. Further, we need stronger preconditions in t_1 to guarantee noninterference.

In our proof, the interesting part for the write is that statement 2 does not interfere with the precondition of statement 3. As seen below, this holds rather straightforward with the precondition of statement 2.

statement 2 and pre(3) by Wr-r-top and Cons

$\{t_1 \ltimes ([\mathtt{x}=1] \wedge \mathbb{1}_{\mathtt{y}}) \wedge t_2 \ltimes \mathbb{1}_0^{\mathtt{y}} \wedge t_2 \ltimes \mathbb{1}_0^{\mathtt{y}}; [\overline{A}(\mathtt{y})]; [\mathtt{x}=1]\}$

$\{t_1 \ltimes ([\mathtt{x}=1] \wedge \mathbb{1}_{\mathtt{y}}) \wedge t_2 \ltimes \mathbb{1}_0^{\mathtt{y}}\}$

$\mathtt{y} :=^R 1$

$\{t_2 \ltimes \mathbb{1}_0^{\mathtt{y}}; [\overline{A}(\mathtt{y})]; [\mathtt{x}=1]\}$

The intuition behind Wr-r-top is that the writing thread perceives the newest write to x. Before statement 2 this would be the write with value 1. The newly introduced write succeeds even this write and therefore is the newest write to x after the transition. After statement 2 this would be the releasing write with value 2.

The thread view of t_1 after executing statement 2 is also the modification view of that write. Guarantees for lists from this thread view onward thus also must hold from the modification view onward. In Wr-r-top, this is captured by I'_t. Because the modification view is the view after the write, we need the additional precondition $\{t \ltimes I_t\}$ $\mathtt{x} :=_t^R e$ $\{t \ltimes I'_t\}$ to make assertions on lists from this view onward. After statement 2 the modification view of the releasing write maps y to the write with value 1. Because the introduced releasing write is the newest write, the final interval is split in two parts:

1. In the first part, views in the list do not succeed the modification view. Therefore, these views must not occur after an acquiring read. The view fulfilling this after statement 2 is executed is the view δ with $val(\delta(\mathtt{x})) = 1$ and $val(\delta(\mathtt{y})) = 0$.

2. In the second part, the view in the list does succeed the modification view. Therefore, it fulfills I'_t. In the litmus test, this is the view δ with $val(\delta(\mathtt{x})) = 1$ and $val(\delta(\mathtt{y})) = 1$, for which $[\![\mathtt{y}=1]\!]_{\Gamma,\delta} = true$ for any C11 state Γ.

This first part is used by Ld-a-shift to guarantee that if a thread perceives an interval beginning with $\overline{A}$ it can be omitted after the read because such a view cannot be the thread view after that thread acquiringly read an releasing write. This can be seen below in (I), which is used for the sequential reasoning for statement 3. Then we can use this with Ld-shift to derive that either $\mathtt{a} = 0$ or t_2 must perceive $\mathtt{x} = 1$.

(I) by Ld-a-shift and Stable-ld

$\{t_2 \ltimes [\overline{A}(\mathtt{y})]; [\mathtt{x}=1]\}$

$\{t_2 \ltimes [\mathtt{x}=1]\}$

$\mathtt{a} \leftarrow^A \mathtt{y}$

$\{t_2 \ltimes [\mathtt{x}=1]\}$

3 by (I), Ld-shift and Cons

$\{t_2 \ltimes \mathbb{1}_0^{\mathtt{y}}; [\overline{A}(\mathtt{y})]; [\mathtt{x}=1]\}$

$\mathtt{a} \leftarrow^A \mathtt{y}$

$\{\mathtt{a} = 0 \vee t_2 \ltimes [\mathtt{x}=1]\}$

$\{\mathtt{a} = 1 \implies t_2 \ltimes [\mathtt{x}=1]\}$

5.5 Peterson's Algorithm

In this section, we take a look at Peterson's algorithm, a case study for mutual exclusion. It revolves around the write to `turn` being atomic, thereby granting *the other thread* access to the critical section if its flag is set. This achieves mutual exclusion, because the second thread executing swap must perceive the flag which was set before the first swap operation, preventing it from entering the critical section.

In addition to the previously used language constructs, the C11 version of the algorithm requires the atomic operation $x.\mathbf{swap}(v)^{\mathrm{RA}}$. It adds the write after which the swap operation occurs to the *covered* set and synchronizes with it like an acquiring read. No new write can be introduced directly after a write in *covered*. We use the interval assertion C_x to denote that all writes to x in a list are in this *covered* set. With this we use the following rules for swaps and covered writes:

Swap-a-shift

$$\frac{\{t \ltimes I\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{\psi\}}$$

Wr-cvd

$$\frac{\{t \ltimes I\}\ x :=_t e\ \{\psi\}}{\{t \ltimes C_x; I\}\ x :=_t e\ \{\psi\}}$$

Swap-wr

$$\frac{}{\{t \ltimes C_x; \mathbb{1}_x \wedge t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{t' \ltimes C_x; \mathbb{1}^x_v\}}$$

Swap-r

$$\frac{\{t \ltimes I_t\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{t \ltimes I'_t\}}{\{t \ltimes (I_t \wedge C_x; \mathbb{1}_x) \wedge t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{t' \ltimes C_x; \overline{A}^x_v; I'_t\}}$$

In the proof outline in Figure 5.5, all write operations to the shared variable `turn` are swap operations. In such a case, we use the notation $C^x_e \equiv t_0 \ltimes C_x; \mathbb{1}^x_e$ to describe that all threads observe only a single write not in the *covered* set with some value matching e. Note that C^x_x is a relaxed version of this, describing that there is only one write not in $\sigma.covered$, since the expression $x = x$ is trivially true.

This example is special, since we want to prove that no two threads can be in the critical section at the same time, instead of a postcondition. We show this by contradiction, assuming they would be. Such a state must fulfill $pre(6)$ for both threads: $\mathtt{a}_1 \wedge (\neg \mathtt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1)$ and $\mathtt{a}_2 \wedge (\neg \mathtt{a}_1 \vee t_1 \ltimes \mathbb{1}^{\mathtt{turn}}_2)$. We derive $t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1 \wedge t_1 \ltimes \mathbb{1}^{\mathtt{turn}}_2$, which is a contradiction by Theorem 3.13. Thus, our assumption must be false.

Theorem 5.5. *The proof outline for Peterson's algorithm in Figure 5.5 is valid.*

The full proof can be seen in Appendix D.5, here we showcase our new proof rules. The program is identical to [DDDW20] and our proof outline is similar to theirs. The proof outlines look somewhat different, because we only allow negations within extended expressions, but for most of their primitive assertions we construct an assertion with similar meaning. There is also a proof outline for Peterson's algorithm with Piccolo

```
**Init** : $\texttt{flag}_1 := \mathit{false}\,;\,\texttt{flag}_2 := \mathit{false}\,;\,\texttt{turn} := 0\,;\,\texttt{a}_1 := \mathit{false}\,;\,\texttt{a}_2 := \mathit{false};$
 **Thread t₁**
 $\{\neg\texttt{a}_1 \wedge t_1 \ltimes \mathbb{1}_{\texttt{flag}_1} \wedge C^{\texttt{turn}}_{\texttt{turn}} \wedge (\neg\texttt{a}_2 \vee (C^{\texttt{turn}}_1 \wedge t_1 \ltimes C_{\texttt{turn}}; \overline{A}(\texttt{turn}); [\texttt{flag}_2]))\}$
1: $\texttt{flag}_1 := \mathit{true};$
 $\{\neg\texttt{a}_1 \wedge t_1 \ltimes [\texttt{flag}_1] \wedge C^{\texttt{turn}}_{\texttt{turn}} \wedge (\neg\texttt{a}_2 \vee (C^{\texttt{turn}}_1 \wedge t_1 \ltimes C_{\texttt{turn}}; \overline{A}(\texttt{turn}); [\texttt{flag}_2]))\}$
2: $\langle \texttt{turn}.\mathbf{swap}(2)^{\mathrm{RA}}\,;\,\texttt{a}_1 := \mathit{true}\rangle$
 $\{\texttt{a}_1 \wedge (\neg\texttt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\texttt{turn}}_1 \vee t_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1])\}$
 **do**
3:   $\texttt{fl}_1 \leftarrow^{A} \texttt{flag}_2;$
   $\{\texttt{a}_1 \wedge (\neg\texttt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\texttt{turn}}_1 \vee (t_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1] \wedge \texttt{fl}_1))\}$
4:   $\texttt{tu}_1 \leftarrow \texttt{turn}$
   $\{\texttt{a}_1 \wedge (\neg\texttt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\texttt{turn}}_1 \vee (t_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1] \wedge \texttt{fl}_1 \wedge \texttt{tu}_1 \neq 1))\}$
5: **until** $(\neg\texttt{fl}_1 \vee \texttt{tu}_1 = 1)$ **do**
 $\{\texttt{a}_1 \wedge (\neg\texttt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\texttt{turn}}_1)\}$
6: Critical section;
 $\{\mathit{true}\}$
7: $\langle \texttt{flag}_1 :=^{R} \mathit{false}\,;\,\texttt{a}_1 := \mathit{false}\rangle$
 $\{\mathit{true}\}$
```

Figure 5.5: Peterson's algorithm (as in [DDDW20]) and its Proof outline, t_2 is symmetric, with $\texttt{fl}_i, \texttt{tu}_i, \texttt{a}_i \in \mathit{Var}_L$ and $\texttt{flag}_i, \texttt{turn} \in \mathit{Var}_G$ for $i \in \{1, 2\}$

in SRA [LDW23], where the assertions are a little bit shorter, because in SRA, there is no for all but one writes to be in the *covered* set to achieve noninterference when executing the swap operation.

After executing the swap statement in thread t_1, we can derive $\neg\texttt{a}_2 \vee t_1 \ltimes [\texttt{flag}_2 \wedge \texttt{turn} \neq 1]$ from $\neg\texttt{a}_2 \vee (t_1 \ltimes C_{\texttt{turn}}; \overline{A}(\texttt{turn}); [\texttt{flag}_2] \wedge C^{\texttt{turn}}_1)$. $t_1 \ltimes [\texttt{turn} \neq 1]$ holds because t_1 afterwards only sees its own write and $t_1 \ltimes [\texttt{flag}_2]$ is shown below:

statement 2 of t_2

$\{t_1 \ltimes C_{\texttt{turn}}; \overline{A}(\texttt{turn}); [\texttt{flag}_2]\}$

$\{t_1 \ltimes \overline{A}(\texttt{turn}); [\texttt{flag}_2]\}$ Wr-cvd

$\{t_1 \ltimes [\texttt{flag}_2]\}$ Swap-a-shift

$\texttt{turn}.\mathbf{swap}(2)^{\mathrm{RA}}$

$\{t_1 \ltimes [\texttt{flag}_2]\}$ Stable-wr

Further, the operation $\texttt{turn}.\mathbf{swap}(2)^{\mathrm{RA}}$ of the other thread might interfere with some operations. On the left, we show that afterwards any thread t_i still perceives only one write not in covered after the statement is executed. On the right, we show that the assertion $t_1 \ltimes C_{\texttt{turn}}; \overline{A}(\texttt{turn}); [\texttt{flag}_2]$ can be derived afterwards.

$t_i \ltimes C_{\texttt{turn}}; \mathbb{1}^{\texttt{turn}}_1$ by Swap-wr

$\{t_2 \ltimes C_{\texttt{turn}}; \mathbb{1}_{\texttt{turn}} \wedge t_i \ltimes C_{\texttt{turn}}; \mathbb{1}_{\texttt{turn}}\}$

$\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$

$\{t_i \ltimes C_{\texttt{turn}}; \mathbb{1}^{\texttt{turn}}_1\}$

$t_1 \ltimes C_t; \overline{A}(\texttt{turn}); [\texttt{flag}_2]$ by Swap-r

$\{C^{\texttt{turn}}_1 \wedge t_2 \ltimes [\texttt{flag}_2]\}$

$\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$

$\{t_1 \ltimes C_t; \overline{A}(\texttt{turn}); [\texttt{flag}_2]\}$

6 Discussion and Related Work

In this chapter, we put our results in the context of related and possible future works. Section 6.1 compares our proof calculus with Piccolo for different memory models. Section 6.2 describes alternative definitions, which future work might benefit from. Section 6.3 discusses more related work and Section 6.4 summarizes our results.

6.1 Unifying Weak Memory Verification with Piccolo

Piccolo was originally a program logic for SRA [LDW23]. Its success led to its extension to TSO and SC [BDW24] and in this thesis we adapted it to C11 RAR. In this section we will compare our adaptation with Piccolo for other weak memory models. We do this by matching assertions with similar meanings or intentions. Using these matched assertions, the proof rules are (syntactically) similar.

Writing thread x.tid This is an extended expression which evaluated to the thread id of the thread having written that write. The C11 semantics do not store this information, but could be easily extended to do so. We argue that in their proof rules where the writing thread is guaranteed, doing the same in C11 would be straightforward. While none of our proof rules take advantage of this, it might be useful to differentiate writes with the same value by different threads, which could help with noninterference proofs.

View-maximality $t \uparrow x$ denotes that t is view-maximal with regards to x. We argue that this is similar to $t \ltimes \mathbb{1}_x$ in our logic, describing that t must observe only the newest write to x.

Proof Rule Soundness If we syntactically replace $t \uparrow x$ with $t \ltimes \mathbb{1}_x$ and ignore $x.tid$, we observe that for every rule other than MCA and St-Other2 we find an equivalent or stronger rule for C11 in this thesis. Most rules are similar, but their rule St-Own is a weaker version of our rule Wr-own-1wr, identical to it for $I \equiv [false]$. MCA is obviously unsound because C11 is not multicopy-atomic and St-Other2 must not hold because it allows deriving $t_2 \ltimes [y \neq 1]; [x = 1]$ as $pre(3)$ in message passing (Figure 5.4), which does not hold for C11.

One Write $\mathbb{1}_x$ is one of our novel interval assertions, which we need for example for the RRC litmus test in Section 5.3, because we need to reason that there is only one write with a certain value in an interval. We presume this assertion could also be helpful with other weak memory models like TSO, where a newly introduced write also may be added to the shared memory before previously introduced writes.

Release-Acquire $\overline{A}(x)$ allows us to assert that synchronization depends on an acquiring read. The other memory models Piccolo currently supports have only a single write type, and therefore, do not require such a differentiation.

Compare and Swap For the swap operation we introduced C_x, which we use similar to $R(x)$ in [LDW23] to describe all but the latest write to a variable. Because of this, we presume it may be possible to unify C_x and $R(x)$.

Altogether, we have adapted Piccolo to C11, and our assertions are generally compatible or more fine-grained (in the case of $\mathbb{1}_x$). With this, our work adds an additional memory model which can be reasoned about with Piccolo.

6.2 Alternative Definitions

In this section, we describe how some of our definitions might be changed to allow stronger conclusions or a more efficient validity computation.

Stronger definition for $\mathbf{t_0}$ The definition below is stronger than the previous shortcut and might allow deriving that different threads must perceive writes in the same order, when t_0 is used in a proof rule like any other thread.

$$\Gamma \vDash t_0 \ltimes I \quad \text{iff } \Gamma, L \vDash I \text{ for every } L \in \sigma.\mathit{Lists}(\mathit{tview}_{\mathbf{Init}})$$

Efficient Validity Computation To show that an assertion $t \ltimes I_1; I_2$ holds in a state, we need to show that for each list $L \in \sigma.\mathit{Lists}(t)$ there exists a split of L such that the interval assertions I_1 and I_2 hold on the parts. Lemma 4.4 states that each view has a specific assertion which always holds for that view. Thus, the definition below could lead to an equivalent validity definition, where we only need to find a split for a single set. This might allow faster automatic validity computation at the cost of a more complex definition.

$$\begin{aligned}
\Gamma, V &\vDash I_1; I_2 && \text{iff } \Gamma, V_1 \vDash I_1 \text{ and } \Gamma, V_2 \vDash I_2 \text{ for some } V_1 \uplus V_2 = V \\
& && \quad \text{with } \neg\exists \delta_1 \in V_1, \delta_2 \in V_2 : \delta_1 \succeq \delta_2 \\
\Gamma &\vDash t \ltimes I && \text{iff } \Gamma, \sigma.\mathit{Views} \vDash I
\end{aligned}$$

6.3 Related Work

In this section, we give an overview of related works both in the context of C11 and memory-model-generic reasoning.

C11 Previous logics reasoning about C11 mostly used separation logic (RSL [VN13], GPS [TVD14, HVQF16], FSL [DV16], FSL++ [DV17], λ_{RN} and iGPS [KDD+17], GPS+ [HVQF18, HQF18], and GPS++ [HQX20]) which require many additional rules for handling ownership of state. These logics handle increasing subsets of C11, beginning with release and acquire accesses and later also fences and release-sequences. Owicki-Gries reasoning for the RA (release-acquire) fragment of C11 was first introduced in [LV15], albeit with a strengthened noninterference check.

Out of thin air reads are a known problem of the C11 memory model. To improve reasoning and this problem, stronger memory models were introduced, e. g. SRA [LGV16], RA+NA [KDD+17], a promising semantics [KHL+17, SPD+18], and RC11 [LVK+17], with the former three being operational and RC11 axiomatic. [DDWD19] developed an operational semantics equivalent to the RAR-fragment of RC11 [LVK+17], which includes releasing, acquiring and relaxed operations. [DDDW20] introduces Owicki-Gries reasoning for an adapted version of this semantics. A solution to allow this behavior is by using an event-structure, allowing reordering certain statements [WBD21]. [WDBD23] builds an equivalent operational semantics based on [DDDW20]. While this mostly follows the Owicki-Gries reasoning style, it allows a partial program order, which results in many additional proof obligations in their program logic.

Another attempt to verify the correctness is model checking [AAAK19, AAJN18]. [AAAK19] also shows that the reachability under RA semantics is undecidable.

Additionally, many of these are (partially) mechanized in theorem provers like Coq [Tea24] and Isabelle [Pau94]. In contrast, our proofs are purely formal, and we only use a semantics mechanized in Isabelle [DDDW20].

Memory-Model-Generic Reasoning We described in Section 6.1, that Piccolo can be used to reason on multiple weak memory models, and we extended this to C11 RAR. As such, this is a contribution to a broader effort to reason about multiple weak memory models at the same time. A similar approach this is reasoning about a generic weak memory model.

[DDDW22] describes a way to verify the validity of a proof outline for multiple weak memory models at the same time by using *weakest liberal preconditions*. [BW23] lifts these to the higher level of program constructs, allowing shorter proofs. [LDW23] introduces a program semantic, which is parametric with regards to the memory model, and the reasoning language Piccolo. It has been extended to reason about deadlocks [DLW24] and multiple weak memory models at the same time [BDW24].

Other verification techniques independent of the memory model include storing written values in *pythia* variables [AC17], a technique to systematically explore possible executions [KRV19], bounded model checking with the memory model as input [dLFHM18, GdLF$^+$19], verifying reordered programs depending on the memory model [Col21] and checking *reordering interference freedom* for statements with assertions in the same thread [CWS21, CWS23].

6.4 Conclusion

In this thesis, we adapted the Piccolo proof calculus to C11 RAR. For this we defined assertions on C11 states, introduced proof rules and proved their soundness. We then used our proof calculus to reason on litmus tests and Peterson's algorithm. This brings a simple and composable assertion language to C11 RAR and extends Piccolo to an additional memory model.

As future work we see the extension of Piccolo to more memory models and unification of our assertions with established ones. In the context of C11, possible avenues include a mechanization of our proof rules and proofs, additional proof rules for release sequences, and handling larger subsets than RAR.

Bibliography

[AAAK19] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankara Narayanan Krishna. Verification of programs under the release-acquire semantics. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1117–1132. ACM, 2019.

[AAJN18] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA):135:1–135:29, 2018.

[AC17] Jade Alglave and Patrick Cousot. Ogre and Pythia: an invariance proof method for weak consistency models. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 3–18. ACM, 2017.

[AdBO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Texts in Computer Science. Springer, 2009.

[BDW24] Lara Bargmann, Brijesh Dongol, and Heike Wehrheim. Unifying weak memory verification using potentials. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I*, volume 14933 of *Lecture Notes in Computer Science*, pages 519–537. Springer, 2024.

[BW23] Lara Bargmann and Heike Wehrheim. Lifting the reasoning level in generic weak memory verification. In Paula Herber and Anton Wijs, editors, *iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands,* *November 13-15, 2023, Proceedings*, volume 14300 of *Lecture Notes in Computer Science*, pages 175–192. Springer, 2023.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.

[Col21] Robert J. Colvin. Parallelized sequential composition and hardware weak memory models. In Radu Calinescu and Corina S. Pasareanu, editors, *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*, volume 13085 of *Lecture Notes in Computer Science*, pages 201–221. Springer, 2021.

[CWS21] Nicholas Coughlin, Kirsten Winter, and Graeme Smith. Rely/guarantee reasoning for multicopy atomic weak memory models. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 292–310. Springer, 2021.

[CWS23] Nicholas Coughlin, Kirsten Winter, and Graeme Smith. Compositional reasoning for non-multicopy atomic architectures. *Formal Aspects Comput.*, 35(2):8:1–8:30, 2023.

[DDDW20] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. Owicki-gries reasoning for C11 RAR. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 11:1–11:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[DDDW22] Simon Doherty, Sadegh Dalvandi, Brijesh Dongol, and Heike Wehrheim. Unifying operational weak memory verification: An axiomatic approach. *ACM Trans. Comput. Log.*, 23(4):27:1–27:39, 2022.

[DDWD19] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. Verifying C11 programs operationally. In Jeffrey K. Hollingsworth and Idit Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 355–365. ACM, 2019.

[dLFHM18] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC with memory models as modules. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[DLW24] Brijesh Dongol, Ori Lahav, and Heike Wehrheim. *A Rely-Guarantee Framework for Proving Deadlock Freedom Under Causal Consistency*, pages 88–108. Springer Nature Switzerland, Cham, 2024.

[DV16] Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 413–430. Springer, 2016.

[DV17] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 448–475. Springer, 2017.

[GdLF$^+$19] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 355–365. Springer, 2019.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HQF18] Mengda He, Shengchao Qin, and João F. Ferreira. Towards a program logic for C11 release-sequences. In Jun Pang, Chenyi Zhang, Jifeng He, and Jian Weng, editors, *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*, pages 28–35. IEEE Computer Society, 2018.

[HQX20] Mengda He, Shengchao Qin, and Zhiwu Xu. A program logic for reasoning about C11 programs with release-sequences. *IEEE Access*, 8:173874–173903, 2020.

[HVQF16] Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. Reasoning about fences and relaxed atomics. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 520–527. IEEE Computer Society, 2016.

[HVQF18] Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. GPS+: reasoning about fences and relaxed atomics. *Int. J. Parallel Program.*, 46(6):1157–1183, 2018.

[ISO11] ISO/IEC 9899:2011. Programming languages — C. International standard, 2011.

[KDD$^+$17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[KHL$^+$17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189. ACM, 2017.

[KRV19] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 96–110. ACM, 2019.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[LDW23] Ori Lahav, Brijesh Dongol, and Heike Wehrheim. Rely-guarantee reasoning for causally consistent shared memory. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I*, volume 13964 of *Lecture Notes in Computer Science*, pages 206–229. Springer, 2023.

[LGV16] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 649–662. ACM, 2016.

[LV15] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2015.

[LVK^{+}17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 618–632. ACM, 2017.

[Mos12] Ben C. Moszkowski. A complete axiom system for propositional interval temporal logic with infinite time. *Log. Methods Comput. Sci.*, 8(3), 2012.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009.

[Pau94] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[SPD^{+}18] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. A separation logic for a promising semantics. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 357–384. Springer, 2018.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[Tea24] The Coq Development Team. *The Coq Reference Manual*, 2024. Release 8.20.0. URL: https://coq.inria.fr/.

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 691–707. ACM, 2014.

[VN13] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for C11 concurrency. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 867–884. ACM, 2013.

[WBD21] Daniel Wright, Mark Batty, and Brijesh Dongol. Owicki-gries reasoning for C11 programs with relaxed dependencies. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2021.

[WDBD23] Daniel Wright, Sadegh Dalvandi, Mark Batty, and Brijesh Dongol. Mechanised operational reasoning for C11 programs with relaxed dependencies. *Formal Aspects Comput.*, 35(2):10:1–10:27, 2023.

A Axioms and Proof Rules

The following axioms and proof rules were used in this thesis.

Compositional rules allow composing statements.

$$\textsc{Seq}\ \frac{\{p\}\ C_1\ \{r\} \qquad \{r\}\ C_2\ \{q\}}{\{p\}\ C_1\ ;C_2\ \{q\}} \qquad \textsc{If}\ \frac{\{p \wedge b\}\ C_1\ \{q\} \qquad \{p \wedge \neg b\}\ C_2\ \{q\}}{\{p\}\ \mathbf{if}\, b\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2\, \mathbf{fi}\ \{q\}}$$

$$\textsc{While}\ \frac{\{p \wedge b\}\ C\ \{p\}}{\{p\}\ \mathbf{while}\, b\, \mathbf{do}\, C\, \mathbf{od}\ \{p \wedge \neg b\}} \qquad \textsc{Until}\ \frac{\{p\}\ C\ \{r\} \qquad \{r\}\ \mathbf{while}\, \neg b\, \mathbf{do}\, C\, \mathbf{od}\ \{r \wedge b\}}{\{p\}\ \mathbf{do}\, C\, \mathbf{until}\, b\, \mathbf{od}\ \{r \wedge b\}}$$

$$\textsc{Parallel}\ \frac{\text{Proof outlines } \{p_i\}\ C_i\ \{q_i\} \text{ are interference free}}{\{\bigwedge_{i=1}^{n} p_i\}\ C_1 \| \ldots \| C_n\ \{\bigwedge_{i=1}^{n} q_i\}} \qquad \textsc{Aux}\ \frac{\{p\}\ C\ \{q\} \qquad vars(q) \cap V = \emptyset}{\{p\}\ C_0\ \{q\}}$$

Logical Rules describe how logical conclusions can be applied.

$$\textsc{Cons}\ \frac{p \Rightarrow p' \qquad \{p'\}\ C\ \{q'\} \qquad q' \Rightarrow q}{\{p\}\ C\ \{q\}} \qquad \textsc{Conj}\ \frac{\{p_1\}\ C\ \{q_1\} \qquad \{p_2\}\ C\ \{q_2\}}{\{p_1 \wedge p_2\}\ C\ \{q_1 \wedge q_2\}}$$

$$\textsc{Disj1}\ \frac{\{p_1\}\ C\ \{q\} \qquad \{p_2\}\ C\ \{q\}}{\{p_1 \vee p_2\}\ C\ \{q\}} \qquad \textsc{Disj2}\ \frac{\{p_1\}\ C\ \{q_1\} \qquad \{p_2\}\ C\ \{q_2\}}{\{p_1 \vee p_2\}\ C\ \{q_1 \vee q_2\}}$$

Basic Rules preserve assertions unrelated to the statements.

$$\textsc{Subst-asgn}\ \frac{}{\{\varphi(r := e)\}\ r :=_t e\ \{\varphi\}} \qquad \textsc{Stable-wr}\ \frac{x \notin fv(\varphi)}{\{\varphi\}\ x :=_t e\ \{\varphi\}} \qquad \textsc{Stable-ld}\ \frac{r \notin fv(\varphi)}{\{\varphi\}\ r \leftarrow_t x\ \{\varphi\}}$$

Load rules are applied to loads.

$$\textsc{Ld-shift}\quad \dfrac{\{t \ltimes I\}\ r \leftarrow_t x\ \{\psi\}}{\{t \ltimes [e(r := x)]; I\}\ r \leftarrow_t x\ \{e \vee \psi\}} \qquad \textsc{Ld-single}\quad \dfrac{}{\{t \ltimes [e(r := x)]\}\ r \leftarrow_t x\ \{e\}}$$

Write rules are applied to writes.

$$\textsc{Wr-own-1wr}\quad \dfrac{}{\{t \ltimes \mathbb{1}_x; I\}\ x :=_t e\ \{t \ltimes \mathbb{1}^x_e; I\}} \qquad \textsc{Wr-top}\quad \dfrac{}{\{t \ltimes \mathbb{1}_x \wedge t' \ltimes I\}\ x :=_t e\ \{t' \ltimes I; \mathbb{1}^x_e\}}$$

$$\textsc{Wr-1wr-single}\quad \dfrac{}{\{t' \ltimes \mathbb{1}_x \wedge I\}\ x :=_t e\ \{t' \ltimes I; \mathbb{1}^x_e\}}$$

$$\textsc{Wr-1wr}\quad \dfrac{\{t' \ltimes I_2\}\ x :=_t e\ \{t' \ltimes I_2'\}}{\{t' \ltimes (I_1 \wedge \mathbb{1}_x); I_2\}\ x :=_t e\ \{t' \ltimes ((I_1; \mathbb{1}^x_e; I_2) \vee (I_1; I_2'))\}}$$

Release-acquire rules handle releasing writes and acquiring reads. Rules for relaxed writes and reads may also be used for them.

$$\textsc{Ld-a-shift}\quad \dfrac{\{t \ltimes I\}\ r \leftarrow^A_t x\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ r \leftarrow^A_t x\ \{\psi\}} \qquad \textsc{Wr-r-top}\quad \dfrac{\{t \ltimes I_t\}\ x :=^R_t e\ \{t \ltimes I_t'\}}{\{t \ltimes (I_t \wedge \mathbb{1}_x) \wedge t' \ltimes I\}\ x :=^R_t e\ \{t' \ltimes I; \overline{A}^x_e; I_t'\}}$$

Swap rules concern themselves with the swap operation and covered writes. Rules for relaxed and releasing writes may also be used for swap operations.

$$\textsc{Swap-a-shift}\quad \dfrac{\{t \ltimes I\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{\psi\}} \qquad \textsc{Wr-cvd}\quad \dfrac{\{t \ltimes I\}\ x :=_t e\ \{\psi\}}{\{t \ltimes C_x; I\}\ x :=_t e\ \{\psi\}}$$

$$\textsc{Swap-wr}\quad \dfrac{}{\{t \ltimes C_x; \mathbb{1}_x \wedge t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{t' \ltimes C_x; \mathbb{1}^x_v\}}$$

$$\textsc{Swap-r}\quad \dfrac{\{t \ltimes I_t\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{t \ltimes I_t'\}}{\{t \ltimes (I_t \wedge C_x; \mathbb{1}_x) \wedge t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)^{\mathrm{RA}}_t\ \{t' \ltimes C_x; \overline{A}^x_v; I_t'\}}$$

B Proofs

In this chapter, we include proofs for statements and lemmas from the thesis other than soundness proofs for proof rules and validity proofs for proof outlines, which are given in Appendix C and Appendix D respectively.

B.1 Well Formedness of Global States

We show the definition of well-formedness again for reference:

$$\begin{aligned} \mathit{wfs}(\sigma) \Leftrightarrow & \mathit{ran}((\bigcup_t \sigma.\mathit{tview}_t) \cup (\bigcup_w \sigma.\mathit{mview}_w)) \subseteq \sigma.\mathit{writes} \wedge \\ & \mathit{finite}(\sigma.\mathit{writes}) \wedge \sigma.\mathit{covered} \subseteq \sigma.\mathit{writes} \wedge \\ & (\forall w : w \in \sigma.\mathit{writes} \Rightarrow \sigma.\mathit{mview}_w(\mathit{var}(w)) = w) \end{aligned}$$

With respect to this definition, Lemma 2.2 states that every reachable global state σ is well-formed, i. e. $\mathit{wfs}(\sigma)$ holds.

Proof. By structural induction.

Initial State:

$$\mathit{ran}((\bigcup_t \sigma.\mathit{tview}_t) \cup (\bigcup_w \sigma.\mathit{mview}_w)) \subseteq \sigma.\mathit{writes}_{\mathbf{Init}}$$

holds because

$$\begin{aligned} & \forall t \in \mathit{Tid}, w \in \mathit{writes}, x_i \in \mathit{Var}_G : \\ & \quad \sigma.\mathit{tview}_t(x_i) = \sigma.\mathit{mview}_w(x_i) = \mathit{view}_{\mathbf{Init}}(x_i) = ((x_i, k_i), 0) \in \sigma.\mathit{writes}_{\mathbf{Init}} \end{aligned}$$

$\mathit{finite}(\sigma.\mathit{writes}_{\mathbf{Init}})$ is trivial. $\sigma.\mathit{covered}_{\mathbf{Init}} \subseteq \sigma.\mathit{writes}_{\mathbf{Init}}$ holds because $\sigma_{\mathbf{Init}}.\mathit{covered} =$

$\emptyset$. $\forall w : w \in \sigma.writes_{\mathbf{Init}} \Rightarrow \sigma.mview_w(var(w)) = w$ holds by the definition of $view_{\mathbf{Init}}$. With all of this, $\mathit{wfs}(\sigma_{\mathbf{Init}})$ holds.

Induction on Read:

$$ran((\bigcup_t \sigma.tview_t) \cup (\bigcup_w \sigma.mview_w)) \subseteq \sigma.writes$$

holds because $\forall x : tview'_t(x) = \sigma.tview_t(x) \vee tview'_t(x) = \sigma.mview_{(w,q)}(x)$ and both of their ranges are in $\sigma.writes$ by induction. $finite(\sigma.writes)$ and $\sigma.covered \subseteq \sigma.writes$ holds by induction because none of the sets are changed. $\forall w : w \in \sigma.writes \Rightarrow \sigma.mview_w(var(w)) = w$ also holds by induction. With this, $\mathit{wfs}(\sigma[tview_t := tview'_t])$ holds.

Induction on Write:

$$ran((\bigcup_t \sigma.tview_t) \cup (\bigcup_w \sigma.mview_w)) \subseteq \sigma.writes$$

holds because $\forall x : tview'_t(x) \in \{\sigma.tview_t(x), (a, q')\}$. $\sigma.tview_t(x) \in writes$ by induction and $(a, q') \in writes' = \sigma.writes \cup \{(a, q')\}$ is trivial. $finite(\sigma.writes)$ and $\sigma.covered \subseteq \sigma.writes$ holds by induction because only one write was added to $\sigma.writes$ and $\sigma.covered$ is unchanged (and $writes' \supseteq \sigma.writes$). $\forall w : w \in \sigma.writes \Rightarrow \sigma.mview_w(var(w)) = w$ also holds by induction for every $\sigma.mview_w$. $tview'_t(var((a, q'))) = (a, q')$, which is the only new $mview$, holds because $var(a) = x$. With this, $\mathit{wfs}(\sigma[tview_t := tview'_t, mview_{(a,q')} := tview'_t, writes := writes'])$ holds.

Induction on Update:

$$ran((\bigcup_t \sigma.tview_t) \cup (\bigcup_w \sigma.mview_w)) \subseteq \sigma.writes$$

holds because $\forall x : tview'_t(x) \in \{\sigma.tview_t(x), \sigma.mview_{(a,q')}(x), (a, q')\}$. $\sigma.tview_t(x)$, $\sigma.mview_{(a,q')}(x) \in writes$ by induction and $(a, q') \in writes' = \sigma.writes \cup \{(a, q')\}$ is trivial. $finite(\sigma.writes)$ and $\sigma.covered \subseteq \sigma.writes$ holds by induction because only one write was added to $\sigma.writes$ and the write added to $\sigma.covered$ by induction is in writes (because it is in $ran(\sigma.mview_{(w,q)})$ as $\sigma.mview_{(w,q)}((w, q)) = (w, q)$ by induction). $\forall w : w \in \sigma.writes \Rightarrow \sigma.mview_w(var(w)) = w$ also holds by induction for every $\sigma.mview_w$. $tview'_t(var((a, q'))) = (a, q')$, which is the only new $mview$, holds because $var(a) = x$. With this, $\mathit{wfs}(\sigma[tview_t := tview'_t, mview_{(a,q')} := tview'_t, writes := writes', covered := covered'])$ holds.

□

B.2 Sublist Validity

In this section, we prove Lemma 3.7 which states the following: For a C11 state Γ and a list L of views, when $\Gamma, L \vDash I$, then $\Gamma, L|_i^j \vDash I$ for every $0 \leq i-1 \leq j \leq n$ (This may be the empty list for $i = j+1$).

Proof. By Induction over the formula we show that $\Gamma, L \vDash I$ implies $\Gamma, L_i^j \vDash I$ for every i, j with $0 \leq i-1 \leq j \leq |L|$:

1. Induction base for $[B]$: Assume $\Gamma, L \vDash [B]$. By definition $[\![B]\!]_{\Gamma,\delta} = true$ for every $\delta \in L$. Then $[\![B]\!]_{\Gamma,\delta} = true$ also holds for every $\delta \in L|_i^j$. By definition $\Gamma, L|_i^j \vDash [B]$ holds.

2. Induction base for $\mathbb{1}_x$: Assume $\Gamma, L \vDash \mathbb{1}_y$ for some $x \in Var_G$. By definition $\delta(x) = \delta'(x)$ for every $\delta, \delta' \in L$. Then $\delta(x) = \delta'(x)$ also holds for every $\delta, \delta' \in L|_i^j$. By definition $\Gamma, L|_i^j \vDash \mathbb{1}_x$ holds.

3. Induction for $I_1; I_2$: Assume $\Gamma, L \vDash I_1; I_2$. By definition, there exists some $0 \leq k \leq |L|$ such that $\Gamma, L|_1^k \vDash I_1$ and $\Gamma, L|_{k+1}^{|L|} \vDash I_2$. We can derive $\Gamma, L|_i^{min(k,j)} \vDash I_1$ and $\Gamma, L|_{max(k+1,i)}^j \vDash I_2$ by induction, because $L|_i^{min(k,j)}$ is a sublist of $L|_1^k$ and $L|_{max(k+1,i)}^j$ of $L|_{k+1}^{|L|}$ respectively. By definition $\Gamma, L|_i^j \vDash I_1; I_2$ holds.

4. Induction for $I_1 \wedge I_2$: Assume $\Gamma, L \vDash I_1 \wedge I_2$. Then $\Gamma, L \vDash I_1$ and $\Gamma, L \vDash I_2$ by definition. We can derive $\Gamma, L|_i^j \vDash I_1$ and $\Gamma, L|_i^j \vDash I_2$ by induction. By definition $\Gamma, L|_i^j \vDash I_1 \wedge I_2$ holds.

5. Induction for $I_1 \vee I_2$: Assume $\Gamma, L \vDash I_1 \vee I_2$. Then $\Gamma, L \vDash I_1$ or $\Gamma, L \vDash I_2$ by definition. We can derive $\Gamma, L|_i^j \vDash I_1$ or $\Gamma, L|_i^j \vDash I_2$ by induction. By definition $\Gamma, L|_i^j \vDash I_1 \vee I_2$ holds.

We derive $\Gamma, L \vDash I$ implies $\Gamma, L|_i^j \vDash I$. □

B.3 Simplified List definition

For the proof of Lemma 3.9 it is helpful to know that $\preceq$ is transitive, so we prove this first.

Lemma B.1. $\preceq$ *is transitive.*

Proof. *$\preceq$ is transitive on writes.* Let $w_1, w_2, w_3 \in \mathtt{W} \times \mathbb{Q}$ with $w_1 \preceq w_2$ and $w_2 \preceq w_3$. Then $tst(w_1) \leq tst(w_2)$ and $tst(w_2) \leq tst(w_3)$. We can derive $tst(w_1) \leq tst(w_3)$.

$\preceq$ is transitive on Views. Let $\delta_1, \delta_2, \delta_3 \in (Var_G \to (\mathtt{W} \times \mathbb{Q}))$ with $\delta_1 \preceq \delta_2$ and $\delta_2 \preceq \delta_3$. Then $\forall x \in Var_G : \delta_1(x) \preceq \delta_2(x)$ and $\forall x \in Var_G : \delta_2(x) \preceq \delta_3(x)$. We can derive $\forall x \in Var_G : (\delta_1(x) \preceq \delta_2(x) \wedge \delta_2(x) \preceq \delta_3(x))$ and thus $\forall x \in Var_G : \delta_1(x) \preceq \delta_3(x)$. □

With this, we begin the proof of Lemma 3.9, which states the following: For any C11 state σ and $\langle \delta_1, \dots, \delta_n \rangle \in \sigma.\mathit{Views}^*$ holds

$$\forall 1 \leq i \leq j \leq n : \delta \preceq \delta_i \preceq \delta_j \iff (n = 0 \vee \delta \preceq \delta_1) \wedge \forall 1 \leq i < n : \delta_i \preceq \delta_{i+1}$$

Proof. Case $n = 0$ holds because both sides are true since there are no valid i, j.

Case $\Longrightarrow$ *and* $n \geq 1$ holds by using $j = i + 1$ for $\delta_i \preceq \delta_{i+1}$ and $i = j = 1$ for $\delta \preceq \delta_1$.

Case $\Longleftarrow$ *and* $n \geq 1$ holds because $\preceq$ is transitive by Lemma B.1. $\forall 1 \leq i \leq j \leq n : \delta_i \preceq \delta_j$ holds by induction: From $\forall 1 \leq i < n : \delta_i \preceq \delta_{i+1}$ and $\forall 1 \leq i < n : \delta_i \preceq \delta_{i+z}$ for some $z \in \mathbb{N}$, we can derive $\forall 1 \leq i < n : \delta_i \preceq \delta_{i+z+1}$ by transitivity. Then $\forall 1 \leq i \leq n : \delta \preceq \delta_i$ holds by $\delta \preceq \delta_1$ and $\forall 1 \leq i \leq j \leq n : \delta_i \preceq \delta_j$. □

B.4 Exchanged States on Write

In this section, we prove Lemma 4.8, which states the following: For Write and Update transition steps with the C11 states Γ before and Γ' after the transition step, if $L \in \sigma.\mathit{Lists}(t')$ for some thread t' and I is an interval formula, then $\Gamma, L \vDash I$ implies $\Gamma', L \vDash I$.

Proof. Let $\Gamma = (\mathit{lst}, \sigma)$ be the state before and $\Gamma' = (\mathit{lst}', \sigma')$ be the state after the transition step. $\mathit{lst} = \mathit{lst}'$ holds for write transitions. We first need to show $[\![E]\!]_{\Gamma,\delta} = [\![E]\!]_{\Gamma',\delta}$ for any global state δ and extended expression E because it is used in interval assertions. We do so by induction over the formula:

1. $[\![\mathtt{a}]\!]_{\Gamma,\delta} = \mathit{lst}(\mathtt{a}) = \mathit{lst}'(\mathtt{a}) = [\![\mathtt{a}]\!]_{\Gamma',\delta}$

2. $[\![x]\!]_{\Gamma,\delta} = \delta(x) = [\![x]\!]_{\Gamma',\delta}$

3. $$\left[\!\!\left[\overline{A}(x)\right]\!\!\right]_{\Gamma,\delta} = \begin{cases} \mathit{true} & \text{if } \delta(x) \in \mathtt{W}_R \times \mathbb{Q} \wedge \neg(\sigma.\mathit{mview}_{\delta(x)} \preceq \delta) \\ \mathit{false} & \text{otherwise} \end{cases}$$
   $$= \begin{cases} \mathit{true} & \text{if } \delta(x) \in \mathtt{W}_R \times \mathbb{Q} \wedge \neg(\sigma'.\mathit{mview}_{\delta(x)} \preceq \delta) \\ \mathit{false} & \text{otherwise} \end{cases}$$
   $$= \left[\!\!\left[\overline{A}(x)\right]\!\!\right]_{\Gamma',\delta}$$
   $\sigma.\mathit{mview}_{\delta(x)} = \sigma'.\mathit{mview}_{\delta(x)}$ holds because modification views are only introduced for new writes and never changed for existing ones.

4. $[\![\neg e]\!]_{\Gamma,\delta} = \neg\, [\![e]\!]_{\Gamma,\delta} = \neg\, [\![e]\!]_{\Gamma',\delta} = [\![\neg e]\!]_{\Gamma',\delta}$

5. $[\![e_1 \oplus e_2]\!]_{\Gamma,\delta} = [\![e_1]\!]_{\Gamma,\delta} \oplus [\![e_2]\!]_{\Gamma,\delta} = [\![e_1]\!]_{\Gamma',\delta} \oplus [\![e_2]\!]_{\Gamma',\delta} = [\![e_1 \oplus e_2]\!]_{\Gamma',\delta}$
   for $\oplus \in \{+, -, *, \div, \dots, =, \leq, \geq, \dots, \wedge, \vee, \dots\}$

With this we can prove $\Gamma, L \vDash I$ implies $\Gamma', L \vDash I$. For this we may assume $\Gamma, L \vDash I$. Similar to before we prove it by induction over the formula:

1. Induction base for $[E_b]$: Assume $\Gamma, L \vDash [E_b]$. By definition $[\![E_b]\!]_{\Gamma,\delta} = \mathit{true}$ for every $\delta \in L$. Then $[\![E_b]\!]_{\Gamma',\delta} = \mathit{true}$ for every $\delta \in L$ because $[\![E]\!]_{\Gamma,\delta} = [\![E]\!]_{\Gamma',\delta}$ for every extended expression E and global state σ. By definition then $\Gamma', L \vDash [E_b]$ holds.

2. Induction base for $\mathbb{1}_x$: Assume $\Gamma, L \vDash \mathbb{1}_x$ for some $x \in Var_G$. By definition $\delta(x) = \delta'(x)$ for every $\delta, \delta' \in L$. By definition then $\Gamma', L \vDash \mathbb{1}_x$ holds.

3. Induction base for C_x: Assume $\Gamma, L \vDash C_x$ for some $x \in Var_G$. By definition $\delta(x) \in \sigma.\mathit{covered}$. Because elements are never removed from *covered*, we can derive $\delta(x) \in \sigma'.\mathit{covered}$. By definition then $\Gamma', L \vDash C_x$ holds.

4. Induction for $I_1; I_2$: Assume $\Gamma, L \vDash I_1; I_2$. By definition there exists some L_1, L_2 with $L_1 \cdot L_2 = L$ such that $\Gamma, L_1 \vDash I_1$ and $\Gamma, L_2 \vDash I_2$. We can derive $\Gamma', L_1 \vDash I_1$ and $\Gamma', L_2 \vDash I_2$ by induction. By definition then $\Gamma', L \vDash I_1; I_2$ holds.

5. Induction for $I_1 \wedge I_2$: Assume $\Gamma, L \vDash I_1 \wedge I_2$. Then $\Gamma, L \vDash I_1$ and $\Gamma, L \vDash I_2$ by definition. We can derive $\Gamma', L \vDash I_1$ and $\Gamma', L \vDash I_2$ by induction. By definition then $\Gamma', L \vDash I_1 \wedge I_2$ holds.

6. Induction for $I_1 \vee I_2$: Assume $\Gamma, L \vDash I_1 \vee I_2$. Then $\Gamma, L \vDash I_1$ or $\Gamma, L \vDash I_2$ by definition. We can derive $\Gamma', L \vDash I_1$ or $\Gamma', L \vDash I_2$ by induction. By definition then $\Gamma', L \vDash I_1 \vee I_2$ holds.

$\square$

C Soundness of Proof Rules

In this section, we include the soundness proofs for the rules we did not prove sound in Chapter 4.

C.1 Basic Rules

In this section, we prove Theorem 4.3, i. e. the soundness of SUBST-ASGN, STABLE-LD and STABLE-WR. For the soundness proofs of the proof rules, let $\Gamma = (lst, \sigma)$ be the C11 state before and $\Gamma' = (lst', \sigma')$ be the C11 state after the transition step. With this we begin the proof of Theorem 4.3.

$$\text{SUBST-ASGN}\ \frac{}{\{\varphi(r := e)\}\ r :=_t e\ \{\varphi\}} \qquad \text{STABLE-LD}\ \frac{r \notin fv(\varphi)}{\{\varphi\}\ r \leftarrow_t x\ \{\varphi\}} \qquad \text{STABLE-WR}\ \frac{x \notin fv(\varphi)}{\{\varphi\}\ x :=_t e\ \{\varphi\}}$$

Proof. SUBST-ASGN
By definition of the transition $\sigma' = \sigma$ and $lst' = lst[r := [\![e]\!]_{\Gamma,\delta}]$ hold. We can prove $\{\varphi(r := e)\}\ t \mapsto r := e\ \{\varphi\}$ by induction over the structure of φ and $\varphi(r := e)$, which are always evaluated the same. The evaluation of e is the same as the evaluation of r after the assignment by the definition of changes by an assignment transition. This holds by induction over the formula. The only interesting case is the evaluation of a (possibly changed) local variable r':

$$[\![r'(r := e)]\!]_{\Gamma,\delta} = \begin{cases} [\![e]\!]_{\Gamma,\delta} & \text{if } r' = r \\ lst(r') & \text{otherwise} \end{cases} = lst'(r') = [\![r']\!]_{\Gamma',\delta}$$

All other evaluations of formulas ($[\![e(r := e)]\!]_{\Gamma,\delta} = [\![e]\!]_{\Gamma',\delta}$) and truth of statements ($\Gamma \vDash \varphi(r := e)$ implies $\Gamma' \vDash \varphi$) are straightforward and will be omitted.

Stable-ld

Proof by induction over the formula. Every $L \in \sigma'.\mathit{Lists}(t)$ is also in $\sigma.\mathit{Lists}(t)$ since $\sigma.\mathit{tview}_t \preceq \sigma'\mathit{tview}_t$ by Theorem 3.1, the set of writes remains unchanged and thus the restrictions are not stronger. The only change in the local state lst is the value of r. Therefore $\Gamma \vDash \varphi$ implies $\Gamma' \vDash \varphi$ holds by induction over the formula, similar to the proof of Subst-asgn.

Stable-wr

By Lemma 4.7 we know for $L = L'[w/w_{\mathit{prev}}]$ that $L \in \sigma.\mathit{Lists}(t')$. We first show $[\![E]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = [\![E]\!]_{\Gamma',\delta'}$ for an extended expression E with $x \notin fv(E)$ because it is used in interval assertions. We do so by induction over the formula:

1. $[\![r]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = \mathit{lst}(r) = \mathit{lst}'(r) = [\![r]\!]_{\Gamma',\delta'}$
2. $[\![y]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = \delta'[w/w_{\mathit{prev}}](y) = \delta'(y) = [\![y]\!]_{\Gamma',\delta'}$ since $x \notin fv(\varphi)$ and thus $y \neq x$
3. $[\![\neg e]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = \neg\, [\![e]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = \neg\, [\![e]\!]_{\Gamma',\delta'} = [\![\neg e]\!]_{\Gamma',\delta'}$
4. $[\![e_1 \oplus e_2]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = [\![e_1]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} \oplus [\![e_2]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = [\![e_1]\!]_{\Gamma',\delta'} \oplus [\![e_2]\!]_{\Gamma',\delta'} = [\![e_1 \oplus e_2]\!]_{\Gamma',\delta'}$
   for $\oplus \in \{+, -, *, \div, \ldots, =, \leq, \geq, \ldots, \wedge, \vee, \ldots\}$

Similarly, we can prove $\Gamma, L \vDash I$ implies $\Gamma', L' \vDash I$ for an interval assertion I with $x \notin fv(I)$. For this we may assume $\Gamma, L \vDash I$. Similar to before we prove it by induction over the formula:

1. Induction base for $[E_b]$: Assume $\Gamma, L'[w/w_{\mathit{prev}}] \vDash [E_b]$. By definition $[\![E_b]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = \mathit{true}$ for every $\delta'[w/w_{\mathit{prev}}] \in L'[w/w_{\mathit{prev}}]$. Then $[\![E_b]\!]_{\Gamma',\delta'} = \mathit{true}$ for every $\delta \in L$ because $[\![E]\!]_{\Gamma,\delta'[w/w_{\mathit{prev}}]} = [\![E]\!]_{\Gamma',\delta'}$. By definition $\Gamma', L' \vDash [E_b]$ holds.
2. Induction base for $\mathbb{1}_y$: Assume $\Gamma, L'[w/w_{\mathit{prev}}] \vDash \mathbb{1}_y$ for some $y \in Var_G$. By definition $\delta(y) = \delta'(y)$ for every $\delta, \delta' \in L'[w/w_{\mathit{prev}}]$. Then $\delta(y) = \delta'(y)$ for every $\delta, \delta' \in L'$ because the values of $y \neq x$ are identical. By definition $\Gamma', L' \vDash \mathbb{1}_y$ holds.
3. Induction for $I_1; I_2$: Assume $\Gamma, L'[w/w_{\mathit{prev}}] \vDash I_1; I_2$. By definition there exists some $0 \leq k \leq |L'|$ such that $\Gamma, L'[w/w_{\mathit{prev}}]|_1^k \vDash I_1$ and $\Gamma, L'[w/w_{\mathit{prev}}]|_k^{|L'|} \vDash I_2$. We can derive $\Gamma', L'|_1^k \vDash I_1$ and $\Gamma', L'|_k^{|L'|} \vDash I_2$ by induction. By definition $\Gamma', L' \vDash I_1; I_2$ holds.
4. Induction for $I_1 \wedge I_2$: Assume $\Gamma, L'[w/w_{\mathit{prev}}] \vDash I_1 \wedge I_2$. Then $\Gamma, L'[w/w_{\mathit{prev}} \vDash I_1$ and $\Gamma, L'[w/w_{\mathit{prev}} \vDash I_2$ by definition. We can derive $\Gamma', L' \vDash I_1$ and $\Gamma', L' \vDash I_2$ by induction. By definition $\Gamma', L' \vDash I_1 \wedge I_2$ holds.
5. Induction for $I_1 \vee I_2$: Assume $\Gamma, L'[w/w_{\mathit{prev}}] \vDash I_1 \vee I_2$. Then $\Gamma, L'[w/w_{\mathit{prev}} \vDash I_1$ or $\Gamma, L'[w/w_{\mathit{prev}} \vDash I_2$ by definition. We can derive $\Gamma', L' \vDash I_1$ or $\Gamma', L' \vDash I_2$ by induction. By definition $\Gamma', L' \vDash I_1 \vee I_2$ holds.

Similarly we can prove $\Gamma \vDash \varphi$ implies $\Gamma' \vDash \varphi$ for an assertion φ with $x \notin fv(\varphi)$. For this we may assume $\Gamma \vDash \varphi$. Similar to before we prove it by induction over the formula:

1. Induction base for $t \ltimes I$: Assume $\Gamma \vDash t \ltimes I$. By definition $\Gamma, L \vDash I$ for every $L \in \sigma.Lists(t)$. Then also $\Gamma, L'[w/w_{prev}] \vDash I$ for every $L' \in \sigma'.Lists(t)$, because $L'[w/w_{prev}] \in \sigma.Lists(t)$ by Lemma 4.7. By definition $\Gamma' \vDash t \ltimes I$ holds.

2. Induction base for e_b: Assume $\Gamma \vDash e_b$. By definition $[\![e_b]\!]_{\Gamma,\delta'[w/w_{prev}]} = true$ for some $\delta \in \sigma.Views$. Then $[\![e_b]\!]_{\Gamma',\delta'} = true$ for some $\delta \in \sigma'.Views$ because $[\![E]\!]_{\Gamma,\delta'[w/w_{prev}]} = [\![E]\!]_{\Gamma',\delta'}$ and $\sigma.Views \subseteq \sigma'.Views$ since $\sigma.writes \subseteq \sigma'.writes$. By definition $\Gamma' \vDash e_b$ holds.

3. Induction for $\varphi \wedge \psi$: Assume $\Gamma \vDash \varphi \wedge \psi$. Then $\Gamma \vDash \varphi$ and $\Gamma \vDash \psi$ by definition. We can derive $\Gamma' \vDash \varphi$ and $\Gamma' \vDash \psi$ by induction. By definition $\Gamma' \vDash \varphi \wedge \psi$ holds.

4. Induction for $\varphi \vee \psi$: Assume $\Gamma \vDash \varphi \vee \psi$. Then $\Gamma \vDash \varphi$ or $\Gamma \vDash \psi$ by definition. We can derive $\Gamma' \vDash \varphi$ or $\Gamma' \vDash \psi$ by induction. By definition $\Gamma' \vDash \varphi \vee \psi$ holds.

With $\Gamma \vDash \varphi$ implies $\Gamma' \vDash \varphi$ we can derive $\{\varphi\}\ x := e\ \{\varphi\}$. □

C.2 Write Rules

For an overview of our general proof idea, we recommend reading Section 4.4.1.

All our newly introduced proof rules derive exactly one Hoare triple with a single statement. For this chapter, we concern ourselves only with some with exactly one write statement. For the soundness proofs of the proof rules, let $\Gamma = (lst, \sigma)$ be the C11 state before and $\Gamma' = (lst', \sigma')$ be the C11 state after the transition.

For the soundness proofs in this section, let $w = (a, q)$ be the write written by the transition. Let w_{prev} be the write previous to w, which is $w_{prev} = max_q\{(a, q) \in \sigma.writes \mid q < tst(w)\}$. This write always exist because all newly introduced writes have a timestamp strictly greater than some other timestamp by *σ.fresh*.

We begin by introducing some lemmas which are useful in multiple soundness proofs, beginning with that write timestamps are unique.

Lemma C.1. *For a global state σ all writes $w \in \sigma.writes$ to a specific shared variable have a unique timestamp:*

$$\forall w, w' \in \sigma.writes : (tst(w) = tst(w') \wedge var(w) = var(w')) \Rightarrow w = w'$$

Proof. Initially there is only one write to each variable, therefore the lemma holds. Every newly introduced write has a unique timestamp q' by *σ.fresh*, because $q' \geq q$ and $q' < tst(w'')$ for every $w'' \in \sigma.writes$ with $tst(w'') > q$. □

Many rules begin some assertion with $\mathbb{1}_x$. To make better use of this, sometimes we insert the thread view of the observing thread at the start of a list. The resulting list is still in the lists of the observing thread by Lemma C.2. By the definition of $\mathbb{1}_x$ we use this to derive that in this interval all writes are $tview_t(x)$.

Lemma C.2. *For $\delta \in \sigma.Views$, if $L \in \sigma.Lists(\delta)$, then also $\langle\delta\rangle \cdot L \in \sigma.Lists(\delta)$.*

Proof. $\delta \in \sigma.Views$ holds by its definition. Let $L' = \langle\delta\rangle \cdot L$ to show where we refer to δ as the first element of the list. $\delta \preceq L'(1) = \delta$ holds because $\delta \preceq \delta$. $L'(1) = \delta \preceq L'(2)$ holds because $\delta \preceq L(1)$. For $1 < i < |L'|$, $\delta \preceq L'(i) \preceq L'(i+1)$ holds because $\delta \preceq L(i-1) \preceq L(i)$. □

With these lemmas we prove Theorem 4.10, the soundness of Wr-own-1wr, Wr-1wr and Wr-1wr-single.

Wr-own-1wr

$$\frac{}{\{t \ltimes \mathbb{1}_x; I\}\ x :=_t e\ \{t \ltimes \mathbb{1}_e^x; I\}}$$

Wr-1wr-single

$$\frac{}{\{t' \ltimes \mathbb{1}_x \wedge I\}\ x :=_t e\ \{t' \ltimes I; \mathbb{1}_e^x\}}$$

Wr-1wr

$$\frac{\{t' \ltimes I_2\}\ x :=_t e\ \{t' \ltimes I_2'\}}{\{t' \ltimes (I_1 \wedge \mathbb{1}_x); I_2\}\ x :=_t e\ \{t' \ltimes ((I_1; \mathbb{1}_e^x; I_2) \vee (I_1; I_2'))\}}$$

$$\mathbb{1}_e^x \equiv \mathbb{1}_x \wedge [x = e]$$

Proof. Wr-own-1wr
Let $L' \in \sigma'.Lists(t)$.

L' can be split at index k with $0 \leq k \leq |L'|$, such that $\delta(x) = w$ for every $\delta \in L'|_1^k$ and $w \notin L'|_{k+1}^{|L|}.writes$. By definition of $\sigma'.Lists(t)$ we know that $\sigma'.tview_t \preceq \delta$ for every $\delta \in L'$. Further any write $L'(i)(x)$ in L' other than w has to have a unique and therefore strictly bigger timestamp by Lemma C.1: $\forall 1 \leq i \leq |L'| : tst(L'(i)(x)) = tst(w) \Rightarrow L'(i)(x) = w$. Therefore, after a write to x other than w, with a strictly greater timestamp, occured in L', w cannot occur because $L'(i) \preceq L'(i+1)$ must hold for every $1 \leq i < |L|$.

By Lemma 4.7 we know $L'[w/w_{prev}] \in \sigma.Lists(t)$. Let $L := \langle\sigma.tview_t\rangle \cdot L'[w/w_{prev}]$. By Lemma C.2 we know that $L \in \sigma.Lists(t)$. With $\Gamma, L \vDash \mathbb{1}_x; I$ we know that L can be split at some index l with $0 \leq l \leq |L|$ such that $\Gamma, L|_1^l \vDash \mathbb{1}_x$ and $\Gamma, L|_{l+1}^{|L|} \vDash I$.

We previously established that all timestamps of writes to x in $L'|_{k+1}^{|L|}$ are greater than $tst(w)$ which in turn is greater than $tst(\sigma.tview_t(x))$ by definition of $\sigma.OW$ and $\sigma.fresh$. Thus $\mathbb{1}_x$ cannot hold for $L|_1^{k+2}$ and we can derive $l \leq k+1$ (The indices are shifted by one because L begins with $\sigma.tview_t$). By Lemma 3.7 we can derive $\Gamma, L'|_{k+1}^{|L'|}[w/w_{prev}] \vDash I$ from $\Gamma, L|_{l+2}^{|L|} \vDash I$ because $L'|_{k+1}^{|L'|} = L|_{k+2}^{|L|}$. Further $L'|_{k+1}^{|L'|}[w/w_{prev}] = L'|_{k+1}^{|L'|}$ because $w \notin L'|_{k+1}^{|L|}.writes$. With Lemma 4.8 we can derive $\Gamma', L'|_{k+1}^{|L'|} \vDash I$.

$\Gamma, L|_1^k \vDash \mathbb{1}_x$ holds trivially by its definition, because $L'|_1^k(x) = w$ for every $1 \leq i \leq |L|$. With $\sigma, L|_{k+1}^{|L|} \vDash I$ and the definition of ; we can derive $\Gamma', L'|_1^k \cdot L'|_{k+1}^{|L|} \vDash \mathbb{1}_x; I$, which is what we needed to show.

Wr-1wr
Let $L' \in \sigma'.\mathit{Lists}(t)$. Then let i, j be integers with $0 \leq i-1 \leq j \leq |L'|$ such that $L(l)(x) = w$ if and only if $i \leq l \leq j$. This is possible because the timestamps of writes to each variable are monotonously increasing (by definition of $\sigma.\mathit{Lists}$) and unique for each write (by Lemma C.1).

Let $L = \langle \sigma.\mathit{tview}_t \rangle \cdot L'[w/w_{prev}]$, with $L \in \sigma.\mathit{Lists}(t)$ by Lemma 4.7 and Lemma C.2.

Let $0 \leq k \leq |L|$ such that $\Gamma, L|_1^k \vDash I_1$, $\Gamma, L|_{k+1}^{|L|} \vDash I_2$ and either $k = |L|$ or $\Gamma, L'' \vDash I_2$ holds for all $L'' \in \sigma.\mathit{Lists}(L(k+1))$. This is possible because if $\Gamma, L|_1^{k+1} \vDash I_1$ does not hold, for any such list L'', $\Gamma, L|_1^{k+1} \cdot L'' \vDash I_1; I_2$ does not hold, but $L|_1^{k+1} \cdot L'' \in \sigma.\mathit{Lists}(t)$.

In the case that $L'.\mathit{writes}$ does not contain w, $L' = L|_2^{|L|}$. From $\Gamma, L \vDash I_1; I_2$ we can derive $\Gamma, L' \vDash I_1; I_2$ by Lemma 3.7 and $\Gamma', L' \vDash I_1; I_2$ by Lemma 4.8. We can derive $\Gamma', L' \vDash I_1; \mathbb{1}_e^x; I_2$ by using the same sublists for I_1 and I_2 as before and the empty list for the interval with $\mathbb{1}_e^x$. Then $\Gamma', L' \vDash ((I_1; \mathbb{1}_e^x; I_2) \vee (I_1; I_2'))$ holds trivially.

Otherwise, there are two cases:

- Case 1, $i \leq k$: We know that $\Gamma, L|_1^i \vDash I_1$ by Lemma 3.7. We can derive $L|_2^i = L'|_1^{i-1}$ because $L'|_1^{i-1}.\mathit{writes}$ does not contain w. By Lemma 3.7, we can derive $\Gamma, L'|_1^{i-1} \vDash I_1$ and by Lemma 4.8 we can derive $\Gamma', L'|_1^{i-1} \vDash I_1$.

  We know that $\Gamma', L'|_i^j \vDash \mathbb{1}_e^x$ by definition of i and j, because $\delta(x) = w$ for every $\delta \in L'|_i^j$.

  We know that $L(1)(x) \preceq w_{prev}$ because it occurs earlier in L, $w_{prev} \prec w$ by definition of w_{prev} and $w \preceq L(j+1)(x)$ because it occurred later in L. We conclude $L(1)(x) \prec L(j+1)(x)$ and therefore $L(1)(x) \neq L(j+1)(x)$. Thus $\Gamma, L|_1^{j+1} \vDash \mathbb{1}_x$ does not hold. By definition of $\Gamma, L \vDash I_1; I_2$ and by Lemma 3.7 we can derive that $\Gamma, L_{j+1}^{|L|} \vDash I_2$ must hold. $\Gamma', L_{j+1}^{|L|} \vDash I_2$ holds by Lemma 4.8.

  Finally with $\Gamma', L'|_1^{i-1} \vDash I_1$, $\Gamma', L'|_i^j \vDash \mathbb{1}_e^x$ and $\Gamma', L_{j+1}^{|L|} \vDash I_2$ we can derive $\Gamma', L \vDash I_1; \mathbb{1}_e^x; I_2$. Then $\Gamma', L' \vDash (I_1; \mathbb{1}_e^x; I_2) \vee (I_1; I_2')$ holds trivially.

- Case 2, $i > k$: Because $L|_2^k = L'|_1^{k-1}$ and $w \notin L'|_1^{k-1}.\mathit{writes}$, $\Gamma', L'|_1^{k-1} \vDash I_1$ holds.

  We know that $k = |L|$ or $\Gamma, L'' \vDash I_2$ holds for all $L'' \in \sigma.\mathit{Lists}(L(k+1))$. If $k = |L|$, the second list is empty and therefore does not contain w. The first part cannot contain w because $i > k$ and before index i the write w does not occur in L'. Thus $L'.\mathit{writes}$ does not contain w, which we already covered earlier.

  Therefore, $\Gamma, L'' \vDash I_2$ must for all $L'' \in \sigma.\mathit{Lists}(L(k+1))$.

  It is possible that there is a thread $t'' \neq t$ with $\sigma.\mathit{tview}_t = L(k+1)$, of which the thread view is not changed by the transition step. For this thread holds $\Gamma \vDash t'' \ltimes I_2$ because $\Gamma, L'' \vDash I_2$ holds and therefore $\Gamma' \vDash t'' \ltimes I_2'$ must hold by the precondition. This allows us to derive $\Gamma', L''' \vDash I_2'$ for every $L''' \in \sigma'.\mathit{Lists}(L(k+1))$.

  We know that $L'|_k^{|L'|} \in \sigma'.\mathit{Lists}(L(k+1))$ because $L(k+1) \preceq L(k+1) = L'|_k^{|L'|}(1)$ and all entries in L' are succeeding each other because $L' \in \sigma.\mathit{Lists}(t')$. Thus we can derive $\Gamma', L'|_k^{|L'|} \vDash I_2'$. With $\Gamma', L'|_1^{k-1} \vDash I_1$ we can conclude $\Gamma', L' \vDash I_1; I_2'$. Then $\Gamma', L' \vDash (I_1; \mathbb{1}_e^x; I_2) \vee (I_1; I_2')$ holds trivially.

In any case we can derive $\Gamma', L' \vDash ((I_1; \mathbb{1}_e^x; I_2) \vee (I_1; I_2'))$.

Wr-1wr-single

Wr-1wr-single is an instance of Wr-1wr with $I_1 \equiv [false]$. The additional requirement holds trivially by Stable-wr. □

C.3 Release-Acquire Rules

In this section, we prove Theorem 4.11, the soundness of Ld-a-shift and Wr-r-top.

Ld-a-shift
$$\frac{\{t \ltimes I\}\ r \leftarrow_t^A x\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ r \leftarrow_t^A x\ \{\psi\}}$$

Wr-r-top
$$\frac{\{t \ltimes I_t\}\ x :=_t^R e\ \{t \ltimes I_t'\}}{\{t \ltimes (I_t \wedge \mathbb{1}_x) \wedge t' \ltimes I\}\ x :=_t^R e\ \{t' \ltimes I; \overline{A}_e^x; I_t'\}}$$

Proof. Ld-a-shift

If $(r \leftarrow^A x, \sigma, lst) \Longrightarrow (\mathbf{skip}, \sigma', lst')$ is possible with an (acquiring) Read transition step, $(r \leftarrow^A x, \sigma', lst) \Longrightarrow (\mathbf{skip}, \sigma', lst')$ is also possible. The only difference between σ and σ' is $tview_t$, the thread view of the writing thread t. $\sigma.tview_t(x)$ is still contained in $\sigma'.OW(t,x)$. Further the thread view after the transition is identical, because repeatedly replacing the value of x to the same read write or applying $\otimes$ multiple times does not change the thread view from the second time onward.

In addition we know that $\sigma.tview_t \preceq \sigma'.tview_t$ by Theorem 3.1. Because the restrictions are weaker, every $L' \in \sigma'.Lists(t)$ is also in $\sigma.Lists(t)$.

Because $\Gamma, L \vDash [\overline{A}(x)]; I$ for every $L \in \sigma.Lists(t)$, we can then derive $\Gamma, L' \vDash [\overline{A}(x)]; I$ for every $L' \in \sigma'.Lists(t)$ (because this is only a subset of the lists). Then by Lemma C.2, $\Gamma, \langle \sigma'.tview_t \rangle \cdot L' \vDash [\overline{A}(x)]; I$.

However, $\Gamma, \langle \sigma'.tview_t \rangle \nvDash [\overline{A}(x)]$ by definition of Read, because $tview_t(x)$ is a nonreleasing write or the thread view was updated using $\otimes$. In the latter case, because $\otimes$ takes the maximum of both inputs for each shared variable, $\sigma'.mview_{\sigma'.tview_t(x)} \preceq \sigma'.tview_t$ holds. In either case we can derive $\Gamma, \langle \sigma'.tview_t \rangle \nvDash [\overline{A}(x)]$.

With $(\sigma', lst), \langle \sigma'.tview_t \rangle \cdot L' \vDash [\overline{A}(x)]; I$ we can derive $\Gamma', \langle \sigma'.tview_t \rangle \cdot L' \vDash I$. With this we can derive $\Gamma', L' \vDash I$. Because we allowed arbitrary L', we know that $\Gamma, L' \vDash I$ for every $L' \in \sigma'.Lists(t)$. Then $(\sigma', lst) \vDash t \ltimes I$ holds.

Because $(\sigma', lst) \Longrightarrow (\sigma', lst')$ is also possible with an acquiring read, $(\sigma', lst) \vDash t \ltimes I$ and we know $\{t \ltimes I\}\ r \leftarrow_t^A x\ \{\psi\}$, we can derive $\Gamma' \vDash \psi$.

Wr-r-top

We begin by showing that the newly introduced write w is the newest write to x. Let $w_{max} := max_q\{(a,q) \in \sigma.writes \mid var(a) = x\}$ be the write to x with the maximal timestamp. Then $\langle \sigma.tview_t, \sigma.tview_t[x := w_{max}] \rangle \in \sigma.Lists(t)$ because $\sigma.tview_t(x) \preceq$

w_{max}. With $t \ltimes \mathbb{1}_x$ we can derive $\sigma.tview_t(x) = w_{max}$.

Write guarantees with $\sigma.fresh$ that the newly introduced write w has an even greater timestamp than the previously latest write: $tst(w) > tst(\sigma.tview_t(var(w)))$. Because it is the only new write ($\sigma'.writes = \sigma.writes \cup \{w\}$), it is succeeds all other writes to x in σ: $tst(w') < tst(w)$ for all $w' \in \sigma'.writes$ with $var(w') = x$. Therefore no other write to x can occur in L' after w in any $L' \in \sigma'.Lists(t')$ of the thread t'.

Let $0 \leq k \leq |L'|$ such that $w \notin L'|_1^k.writes$ and $\delta(x) = w$ for every $\delta \in L'|_{k+1}^{|L'|}$. We know that $L'[w/w_{prev}] \in \sigma.Lists(t)$, therefore also $L'|_1^k[w/w_{prev}] \in \sigma.Lists(t)$. Further by definition of k, $L'|_1^k[w/w_{prev}] = L'|_1^k$ because $w \notin L'|_1^k.writes$. Therefore we can derive $\Gamma, L'|_1^k \vDash I$ and by Lemma 4.8 $\Gamma', L'|_1^k \vDash I$.

$\Gamma', L'|_{k+1}^{|L'|} \vDash \mathbb{1}_x \wedge [x = e]$ holds because $\delta(x) = w$ for every $\delta \in L'|_{k+1}^{|L'|}$. Because $\preceq$ is transitive by Lemma B.1, there exists a $k \leq l \leq |L'|$ splitting $L'|_{k+1}^{|L'|}$ such that $\neg(\sigma'.mview_w \preceq \delta_1)$ holds for every $\delta_1 \in L'|_{k+1}^{l}$ and $\sigma'.mview_w \preceq \delta_2$ holds for every $\delta_2 \in L'|_{l+1}^{|L'|}$.

Then, because w is a releasing write and $\neg(\sigma'.mview_w \preceq \delta_1)$, $\Gamma', L'|_{k+1}^{l} \vDash \overline{A}(x)$ holds. With w being the only write in that list, $\Gamma', L'|_{k+1}^{l} \vDash \overline{A}(x) \wedge \mathbb{1}_e^x$ holds.

Further, because $\sigma'.mview_w \preceq \delta_1$ for every $\delta_2 \in L'|_{l+1}^{|L'|}$ and $\sigma'.mview_w = \sigma'.tview_t$, we know that $L'|_{l+1}^{|L'|} \in \sigma'.Lists(t)$. Therefore, we can derive $\Gamma', L'|_{l+1}^{|L'|} \vDash I'_t$ by the precondition $\{t \ltimes I_t\}\ t \mapsto x :=^R e\ \{t \ltimes I'_t\}$.

As $L'|_1^k \cdot L'|_{k+1}^{l} \cdot L|_{l+1}^{|L'|} = L'$, this is what we needed to show: $\Gamma', L' \vDash I; \overline{A}_e^x; I'_t$. $\square$

C.4 Rules for Perception with Swap

In this section, we prove Theorem 4.12, the soundness of Swap-a-shift and Wr-cvd.

Swap-a-shift

$$\frac{\{t \ltimes I\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{\psi\}}{\{t \ltimes [\overline{A}(x)]; I\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{\psi\}}$$

Wr-cvd

$$\frac{\{t \ltimes I\}\ x :=_t e\ \{\psi\}}{\{t \ltimes C_x; I\}\ x :=_t e\ \{\psi\}}$$

Proof. Swap-a-shift
If $(x.\mathbf{swap}(r)^{\mathrm{RA}}, \sigma, lst) \Longrightarrow (\mathbf{skip}, \sigma', lst')$ is possible with an Update transition step, then $(x.\mathbf{swap}(r)^{\mathrm{RA}}, \sigma'', lst) \Longrightarrow (\mathbf{skip}, \sigma', lst')$ is also possible for

$$\sigma'' = \begin{cases} \sigma[tview_t := \sigma.tview_t \otimes \sigma.mview_w], & \text{if } w \in \mathtt{W}_R \times \mathbb{Q} \\ \sigma[tview_t := \sigma.tview_t[x := w]], & \text{otherwise} \end{cases}$$

where $w := max_q\{(a, q) \in \sigma.writes \mid var(a) = x \wedge q < tst(\sigma'.tview_t(x))\}$ is the write after which the transition inserted $\sigma'.tview_t(x)$.

By definition of OW (Equation (2.2)), w is still contained in $\sigma''.OW(t, x)$. Further the thread view after the transition is identical, because applying $\otimes$ an additional time does not change the resulting thread view.

In addition we know that $\sigma.tview_t \preceq \sigma''.tview_t$ by construction of $\otimes$. Because the restrictions are weaker, every $L' \in \sigma''.Lists(t)$ is also in $\sigma.Lists(t)$.

Because $(\sigma, lst), L \vDash [\overline{A}(x)]; I$ for every $L \in \sigma.Lists(t)$, we can then derive $(\sigma'', lst), L' \vDash [\overline{A}(x)]; I$ for every $L' \in \sigma''.Lists(t)$ (because this is only a subset of the lists). Then by Lemma C.2, $(\sigma'', lst), \langle \sigma''.tview_t \rangle \cdot L' \vDash [\overline{A}(x)]; I$.

However, $(\sigma'', lst), \langle \sigma''.tview_t \rangle \nvDash [\overline{A}(x)]$ by definition of σ'', because w is a nonreleasing write or the thread view was updated using $\otimes$. In the latter case, because $\otimes$ takes the maximum of both inputs for each shared variable, $\sigma.mview_w \preceq \sigma''.tview_t$ holds. In either case we can derive $(\sigma'', lst), \langle \sigma''.tview_t \rangle \nvDash [\overline{A}(x)]$.

With $(\sigma'', lst), \langle \sigma''.tview_t \rangle \cdot L' \vDash [\overline{A}(x)]; I$ we can derive $(\sigma'', lst), \langle \sigma''.tview_t \rangle \cdot L' \vDash I$. With this we can derive $(\sigma'', lst), L' \vDash I$. Because we allowed arbitrary L', we know that $(\sigma'', lst), L' \vDash I$ for every $L' \in \sigma''.Lists(t)$. Then $(\sigma'', lst) \vDash t \ltimes I$ holds.

Because $(\sigma'', lst) \Longrightarrow (\sigma', lst')$ is also possible with an acquiring read, $(\sigma'', lst) \vDash t \ltimes I$ and we know $\{t \ltimes I\}$ $x.\mathbf{swap}(r)^{\mathrm{RA}}$ $\{\psi\}$, we can derive $(\sigma', lst') \vDash \psi$.

Wr-cvd
If $(x :=_t e, \sigma, lst) \Longrightarrow (\mathbf{skip}, \sigma', lst')$ is possible with an Write/Update transition step, then $(x :=_t e, \sigma'', lst) \Longrightarrow (\mathbf{skip}, \sigma', lst')$ is also possible for $\sigma'' = \sigma[tview_t := \sigma.tview_t[x := w]]$ where $w := max_q\{(a, q) \in \sigma.writes \mid var(a) = x \wedge q < tst(\sigma'.tview_t(x))\}$ is the write after which the transition inserted $\sigma'.tview_t(x)$.

By definition of OW, w is still contained in $\sigma''.OW(t, x)$. Further the thread view after the transition is identical, because the write w is the same and the value of other shared variables of views are unchanged.

In addition we know that $\sigma.tview_t \preceq \sigma''.tview_t$ since $w \in \sigma.OW(t, x)$. Because the restrictions are weaker, every $L' \in \sigma''.Lists(t)$ is also in $\sigma.Lists(t)$.

Because $(\sigma, lst), L \vDash C_x; I$ for every $L \in \sigma.Lists(t)$, we can then derive $(\sigma'', lst), L' \vDash C_x; I$ for every $L' \in \sigma''.Lists(t)$ (because this is only a subset of the lists). Then by Lemma C.2, $(\sigma'', lst), \langle \sigma''.tview_t \rangle \cdot L' \vDash C_x; I$.

However, $(\sigma'', lst), \langle \sigma''.tview_t \rangle \nvDash C_x$ by definition of Write/Update, because w is not allowed to be in $\sigma.covered$.

With $(\sigma'', lst), \langle \sigma''.tview_t \rangle \cdot L' \vDash C_x; I$ we can derive $(\sigma'', lst), \langle \sigma''.tview_t \rangle \cdot L' \vDash I$. With this we can derive $(\sigma'', lst), L' \vDash I$. Because we allowed arbitrary L', we know that $(\sigma'', lst), L' \vDash I$ for every $L' \in \sigma''.Lists(t)$. Then $(\sigma'', lst) \vDash t \ltimes I$ holds.

Because $(\sigma'', lst) \Longrightarrow (\sigma', lst')$ is also possible with an acquiring read, $(\sigma'', lst) \vDash t \ltimes I$ and we know $\{t \ltimes I\}$ $x :=_t e$ $\{\psi\}$, we can derive $(\sigma', lst') \vDash \psi$. □

C.5 Rules for Writing with Swap

In this section, we prove Theorem 4.13, the soundness of SWAP-WR and SWAP-R.

SWAP-WR

$$\frac{}{\{t \ltimes C_x; \mathbb{1}_x \wedge t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{t' \ltimes C_x; \mathbb{1}_v^x\}}$$

SWAP-R

$$\frac{\{t \ltimes I_t\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{t \ltimes I'_t\}}{\{t \ltimes (I_t \wedge C_x; \mathbb{1}_x) \wedge t' \ltimes C_x; \mathbb{1}_x\}\ x.\mathbf{swap}(v)_t^{\mathrm{RA}}\ \{t' \ltimes C_x; \overline{A}_v^x; I'_t\}}$$

Proof. Here we show the soundness of SWAP-WR and extend it to a proof of SWAP-WR.

We are concerned with a transition step $(x.\mathbf{swap}(v)_t^{\mathrm{RA}}, \sigma, lst) \Longrightarrow (\mathbf{skip}, \sigma', lst')$. For both rules we know $(\sigma, lst) \vDash t \ltimes C_x; \mathbb{1}_x \wedge t' \ltimes C_x; \mathbb{1}_x$. We can derive that t (and analogous t') must only observe one write $w_{prev} \in \sigma.OW(t,x) \backslash \sigma.covered$ not in covered. Otherwise, if there were two such writes w_1 and w_2, a list $\langle \delta_1, \delta_2 \rangle$ with $\delta_1(x) = w_1$ and $\delta_2(x) = w_2$ would not fulfill $C_x; \mathbb{1}_x$ because neither write is in $\sigma.covered$ and they are not identical. Further, w_{prev} must be the final write to x in σ (i. e. with the maximal timestamp among writes to x) by the same reasoning, since we can assume the final write to occur after w_{prev}.

With this we know that the UPDATE transition step adds w_{prev} to the covered set and introduces a new write $w = \sigma'.tview_t(x)$ with $val(w) = [\![e]\!]_{(\sigma,lst),\sigma.tview_t}$ and $\sigma.fresh(tst(w_{prev}), tst(w))$. By definition of *fresh* (Equation (2.1)), we know that this new write w has the new maximal timestamp among writes to x. Therefore, no other write to x can occur after w in any $L' \in \sigma'.Lists(t)$.

Let $0 \le k \le |L'$ such that $w_{prev} \notin L'|_1^k.writes$ and $\delta(x) = w$ for every $\delta \in L'|_{k+1}^{|L'|}$. We know that $L'[w/w_{prev}] \in \sigma.Lists(t)$ by Lemma 4.7, therefore also $L'|_1^k[w/w_{prev}] \in \sigma.Lists(t)$. Further by definition of k, $L'|_1^k[w/w_{prev}] = L'|_1^k$ because $w \notin L'|_1^k.writes$. Therefore we can derive $(\sigma, lst), L'|_1^k \vDash I$ and by Lemma 4.8 $(\sigma', lst'), L'|_1^k \vDash I$.

$(\sigma', lst'), L'|_{k+1}^{|L'|} \vDash \mathbb{1}_x \wedge [x = e]$ holds trivially because $\delta(x) = w$ for every $\delta \in L'|_{k+1}^{|L'|}$. We can derive $(\sigma', lst'), L'|_1^k \cdot L'|_{k+1}^{|L'|} \vDash I; \mathbb{1}_e^x$. As $L'|_1^k \cdot L'|_{k+1}^{|L'|} = L'$, this is what we needed to show for SWAP-WR: $(\sigma', lst'), L' \vDash I; \mathbb{1}_e^x$.

SWAP-R(continuation)
Because $\preceq$ is transitive by Lemma B.1, there exists a $k \le l \le |L'|$ splitting $L'|_{k+1}^{|L'|}$ such that $\neg(\sigma'.mview_w \preceq \delta_1)$ holds for every $\delta_1 \in L'|_{k+1}^l$ and $\sigma'.mview_w \preceq \delta_2$ holds for every $\delta_2 \in L'|_{l+1}^{|L'|}$.

Then, because w is a releasing write and $\neg(\sigma'.mview_w \preceq \delta_1)$, $(\sigma', lst'), L'|_{k+1}^l \vDash \overline{A}(x)$ holds. With w being the only write in that list, $(\sigma', lst'), L'|_{k+1}^l \vDash \overline{A}(x) \wedge \mathbb{1}_e^x$ holds.

Further, because $\sigma'.mview_w \preceq \delta_1$ for every $\delta_2 \in L'|_{l+1}^{|L'|}$ and $\sigma'.mview_w = \sigma'.tview_t$, we know that $L'|_{l+1}^{|L'|} \in \sigma'.Lists(t)$. Therefore we can derive $(\sigma', lst'), L'|_{l+1}^{|L'|} \vDash I'_t$ by the precondition $\{t \ltimes I_t\}\ t \mapsto x :=^R e\ \{t \ltimes I'_t\}$.

As $L'|_1^k \cdot L'|_{k+1}^l \cdot L'|_{l+1}^{|L'|} = L'$, this is what we needed to show: $(\sigma', lst'), L' \vDash I; \overline{A}_e^x; I'_t$.
$\square$

D Validity of Proof Outlines

In this chapter, we prove the validity of the proof outlines shown in this thesis. We do so by first proving the sequential validity of each single thread by showing $\{pre(C)\}$ C $\{post(C)\}$ for every (instrumented) statement C in that thread. For this we refer to statements in a proof outline by their number to be concise and avoid ambiguity. Additionally, we show noninterference for every statement with every assertion in each other thread.

D.1 Load Buffering

In this section, we prove Theorem 5.1, the validity of the proof outline for load buffering seen below.

$$
\begin{array}{c}
\textbf{Init} : \mathtt{x} := 0\,;\mathtt{x} := 0\,;\mathtt{a} := 0\,;\mathtt{b} := 0;\\
\{t_1 \ltimes [\mathtt{x} = 0] \wedge t_2 \ltimes [\mathtt{y} = 0] \wedge \mathtt{a} = 0 \wedge \mathtt{b} = 0\}\\
\begin{array}{l||l}
\textbf{Thread } \mathbf{t_1} & \textbf{Thread } \mathbf{t_2}\\
\{t_2 \ltimes [\mathtt{y} = 0] \wedge \mathtt{b} = 0\} & \{t_1 \ltimes [\mathtt{x} = 0] \wedge \mathtt{a} = 0\}\\
1 : \mathtt{a} \leftarrow \mathtt{x}; & 3 : \mathtt{b} \leftarrow \mathtt{y};\\
\{t_2 \ltimes [\mathtt{y} = 0] \wedge \mathtt{b} = 0\} & \{t_1 \ltimes [\mathtt{x} = 0] \wedge \mathtt{a} = 0\}\\
2 : \mathtt{y} := 1 & 4 : \mathtt{x} := 1\\
\{\mathtt{a} = 0 \vee \mathtt{b} = 0\} & \{\mathtt{a} = 0 \vee \mathtt{b} = 0\}
\end{array}\\
\{\mathtt{a} = 0 \vee \mathtt{b} = 0\}
\end{array}
$$

Proof. **Thread $\mathbf{t_1}$**

1 by Stable-ld:

$\{t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{b} = 0\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{b} = 0\}$

2 by Stable-wr, Cons:

$\{t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{b} = 0\}$
$\{\mathtt{b} = 0\}$
$\mathtt{y} := 1$
$\{\mathtt{b} = 0\}$
$\{\mathtt{a} = 0 \lor \mathtt{b} = 0\}$

Thread $\mathbf{t_2}$

Because the threads are symmetrical, the proof is analogous to t_1. Only the variable names need to be swapped for their symmetric counterparts.

Noninterference

Statement 1

(I) by Stable-ld:
$\{t_1 \ltimes [\mathtt{x} = 0]\}\mathtt{a} \leftarrow \mathtt{x}\{t_1 \ltimes [\mathtt{x} = 0]\}$

(II) by Ld-single:
$\{t_1 \ltimes [\mathtt{x} = 0]\}\mathtt{a} \leftarrow \mathtt{x}\{\mathtt{a} = 0\}$

stability of pre(3) and pre(4) by (I), (II), Conj and Cons:

$\{t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{b} = 0 \land t_1 \ltimes [\mathtt{x} = 0] \land \mathtt{a} = 0\}$
$\{t_1 \ltimes [\mathtt{x} = 0] \land t_1 \ltimes [\mathtt{x} = 0]\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{t_1 \ltimes [\mathtt{x} = 0] \land \mathtt{a} = 0\}$

stability of post(4) by Stable-ld and Cons:

$\{t_2 \ltimes [\mathtt{y} = 0] \land \mathtt{b} = 0 \land (\mathtt{a} = 0 \lor \mathtt{b} = 0)\}$
$\{\mathtt{b} = 0\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{\mathtt{b} = 0\}$
$\{\mathtt{a} = 0 \lor \mathtt{b} = 0\}$

Statement 2-4 Statement 2 does not interfere with any assertion in Thread t_2 by Stable-wr because none of its assertions contain $\mathtt{y}$. Statements 3 and 4 are analogous to Statements 1 and 2 respectively, only variable names need to be swapped for their symmetric counterparts. □

D.2 RRC2

In this section, we prove Theorem 5.2, the validity of the proof outline for RRC2 seen below.

$$\textbf{Init} : \mathtt{x} := 0;$$
$$\{t_1 \ltimes \mathbb{1}_{\mathtt{x}} \wedge t_2 \ltimes [\mathtt{x} = 0]\}$$

Thread t_1	**Thread t_2**
$\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$	$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 2]\}$
$1 : \mathtt{x} := 1;$	$3 : \mathtt{a} \leftarrow \mathtt{x};$
$\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$	$\{\mathtt{a} = 2 \Rightarrow t_1 \ltimes [\mathtt{x} = 2]\}$
$2 : \mathtt{x} := 2$	$4 : \mathtt{b} \leftarrow \mathtt{x}$
$\{true\}$	$\{\mathtt{a} = 2 \Rightarrow \mathtt{b} = 2\}$

$$\{\mathtt{a} = 2 \Rightarrow \mathtt{b} = 2\}$$

Proof. **Thread t_1**

Statement 1

(I) by Wr-top and Cons:

$\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$
$\mathtt{x} := 1$
$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 1]\}$
$\{t_2 \ltimes [\mathtt{x} \neq 2]\}$

(II) by Wr-own-1wr and Cons:

$\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$
$\{t_1 \ltimes \mathbb{1}_{\mathtt{x}}; [false]\}$
$\mathtt{x} := 1$
$\{t_1 \ltimes \mathbb{1}^{\mathtt{x}}_{1}; [false]\}$
$\{t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$

by (I), (II), Conj and Cons:

$$\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}\mathtt{x} := 1\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$$

Statement 2

by Stable-wr and Cons

$$\{t_2 \ltimes [\mathtt{x} \neq 2] \wedge t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}\{true\}\mathtt{x} := 2\{true\}$$

Thread t_2

Statement 3

By Stable-ld, Ld-shift and Cons

$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 2]\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{a \neq 2 \vee t_2 \ltimes [\mathtt{x} = 2]\}$
$\{a = 2 \Rightarrow t_2 \ltimes [\mathtt{x} = 2]\}$

Statement 4

(I) by LD-SINGLE

$\{t_2 \ltimes [\mathtt{x} = 2]\}\mathtt{b} \leftarrow \mathtt{x}\{\mathtt{b} = 2\}$

(II) by STABLE-LD

$\{a \neq 2\}\mathtt{b} \leftarrow \mathtt{x}\{a \neq 2\}$

by (I), (II), DISJ2 and CONS

$\{a \neq 2 \lor t_2 \ltimes [\mathtt{x} = 2]\}$

$\mathtt{b} \leftarrow \mathtt{x}$

$\{a \neq 2 \lor \mathtt{b} = 2\}$

$\{a = 2 \Rightarrow \mathtt{b} = 2\}$

Noninterference

pre(1), pre(2), post(2), post(4): No statement from t_2 interferes with an assertions of t_1 by STABLE-LD. Statements from t_1 do not interfere with $\mathtt{a} = 2 \implies b = 2$ by STABLE-WR.

pre(4): The following holds for $n \in \{1, 2\}$:

$$t_2 \ltimes [\mathtt{x} \neq 2] \land t_2 \ltimes [\mathtt{x} = 2] \implies \mathit{false} \tag{D.1}$$

by STABLE-WR, CONS and D.1

$\{t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}} \land (t_2 \ltimes [\mathtt{x} = 2] \lor a \neq 2)\}$

$\{(t_2 \ltimes [\mathtt{x} \neq 2] \land t_2 \ltimes [\mathtt{x} = 2]) \lor a \neq 2\}$

$\{a \neq 2\}$

$\mathtt{x} := n$

$\{a \neq 2\}$

$\{a \neq 2 \lor t_2 \ltimes [x = 2]\}$

pre(3)

1 by WR-TOP and CONS

$\left\{\begin{array}{c} t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}} \land \\ t_2 \ltimes [x \neq 2]; [x = 2] \end{array}\right\}$

$\{t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$

$\mathtt{x} := 1$

$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 1]\}$

$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} \neq 2]\}$

$\{t_2 \ltimes [\mathtt{x} \neq 2]\}$

$\{t_2 \ltimes [\mathtt{x} \neq 2]; [x = 2]\}$

2 by WR-TOP and CONS

$\left\{\begin{array}{c} t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}} \land \\ t_2 \ltimes [x \neq 2]; [x = 2] \end{array}\right\}$

$\{t_2 \ltimes [\mathtt{x} \neq 2] \land t_1 \ltimes \mathbb{1}_{\mathtt{x}}\}$

$\mathtt{x} := 2$

$\{t_2 \ltimes [\mathtt{x} \neq 2]; [\mathtt{x} = 2]\}$

□

D.3 RRC

In this section, we prove Theorem 5.3, the validity of the proof outline for RRC seen below.

$$\textbf{Init} : \mathtt{x} := 0;$$
$$\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{0}\}$$

$$
\begin{array}{l||l||l||l}
\textbf{Thread } \mathtt{t_1} & \textbf{Thread } \mathtt{t_2} & \textbf{Thread } \mathtt{t_3} & \textbf{Thread } \mathtt{t_4} \\
\left\{\begin{array}{l}\mathtt{a} \neq 1 \land \\ t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\end{array}\right\} & \left\{\begin{array}{l}\mathtt{c} \neq 2 \land \\ t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{01}\end{array}\right\} & \{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \lor \mathbb{1}^{\mathtt{x}}_{021}\} & \{t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \lor \mathbb{1}^{\mathtt{x}}_{021}\} \\
1 : \mathtt{x} := 1 & 2 : \mathtt{x} := 2 & 3 : \mathtt{a} \leftarrow \mathtt{x}; & 5 : \mathtt{c} \leftarrow \mathtt{x}; \\
\{\mathit{true}\} & \{\mathit{true}\} & \{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\} & \{\mathtt{c} = 2 \Rightarrow t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{21}\} \\
 & & 4 : \mathtt{b} \leftarrow \mathtt{x} & 6 : \mathtt{d} \leftarrow \mathtt{x} \\
 & & \left\{\begin{array}{l}(a = 1 \land b = 2) \\ \quad \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\end{array}\right\} & \left\{\begin{array}{l}(c = 2 \land d = 1) \\ \quad \Rightarrow t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{1}\end{array}\right\}
\end{array}
$$
$$\{(\mathtt{a} = 1 \land \mathtt{b} = 2 \land \mathtt{c} = 2) \Rightarrow \mathtt{d} \neq 1\}$$

Proof. **Threads $\mathtt{t_1}$ and $\mathtt{t_2}$**

by Stable-wr and Cons
$$\{\mathtt{a} \neq 1 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}\{\mathit{true}\}\mathtt{x} := 1\{\mathit{true}\}$$

by Stable-wr and Cons
$$\{\mathtt{c} \neq 2 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{01}\}\{\mathit{true}\}\mathtt{x} := 2\{\mathit{true}\}$$

Threads $\mathtt{t_3}$ and $\mathtt{t_4}$

Thread t_4 is symmetric to t_3. For its proofs, only the variable/ thread names and values need to be swapped with their counterparts.

Statement 3

(I) by Stable-ld, Ld-shift and Cons
$$
\begin{array}{l}
\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012}\} \\
\{t_3 \ltimes [\mathtt{x} = 0]; \mathbb{1}^{\mathtt{x}}_{12}\} \\
\mathtt{a} \leftarrow \mathtt{x} \\
\{\mathtt{a} = 0 \lor t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\} \\
\{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}
\end{array}
$$

(II) by Stable-ld, Ld-shift and Cons
$$
\begin{array}{l}
\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{021}\} \\
\{t_3 \ltimes [\mathtt{x} \neq 1]; \mathbb{1}^{\mathtt{x}}_{1}\} \\
\mathtt{a} \leftarrow \mathtt{x} \\
\{\mathtt{a} \neq 1 \lor t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{1}\} \\
\{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}
\end{array}
$$

by (I), (II) and Disj1
$$
\begin{array}{l}
\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \lor \mathbb{1}^{\mathtt{x}}_{021}\} \\
\mathtt{a} \leftarrow \mathtt{x} \\
\{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}
\end{array}
$$

Statement 4

(I) by Stable-ld, Ld-shift and Cons

$\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}$

$\{t_3 \ltimes [b \neq 2]; \mathbb{1}^{\mathtt{x}}_{2}\}$

$\mathtt{b} \leftarrow \mathtt{x}$

$\{\mathtt{b} \neq 2 \lor t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\}$

by (I), Stable-ld, Disj2 and Cons

$\{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}$

$\{\mathtt{a} \neq 1 \lor t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}$

$\mathtt{b} \leftarrow \mathtt{x}$

$\{\mathtt{a} \neq 1 \lor \mathtt{b} \neq 2 \lor t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\}$

$\{(\mathtt{a} = 1 \land \mathtt{b} = 2) \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\}$

Conclusion Here we prove that the conclusion $(\mathtt{a} = 1 \land \mathtt{b} = 2 \land \mathtt{c} = 2) \implies \mathtt{d} \neq 1$ can be derived from $post(1) \land post(2) \land post(4) \land post(6)$:

$$\begin{aligned}
& true \land true \land ((\mathtt{a} = 1 \land \mathtt{b} = 2) \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}) \land ((\mathtt{c} = 2 \land \mathtt{d} = 1) \Rightarrow t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{1}) \\
\implies & (\mathtt{a} = 1 \land \mathtt{b} = 2 \land \mathtt{c} = 2 \land \mathtt{d} = 1) \implies (t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2} \land t_4 \ltimes \mathbb{1}^{\mathtt{x}}_{1}) \\
\implies & (\mathtt{a} = 1 \land \mathtt{b} = 2 \land \mathtt{c} = 2 \land \mathtt{d} = 1) \implies false \\
\implies & \neg(\mathtt{a} = 1 \land \mathtt{b} = 2 \land \mathtt{c} = 2 \land \mathtt{d} = 1) \\
\implies & (\mathtt{a} = 1 \land \mathtt{b} = 2 \land \mathtt{c} = 2) \implies \mathtt{d} \neq 1
\end{aligned}$$

Noninterference

The threads t_1 and t_2 as well as t_3 and t_4 are symmetric. So are their combinations t_1, t_3 and t_2, t_4 as well as t_1, t_4 and t_2, t_3. The same holds for t_1, t_2 and t_2, t_1. The noninterference proofs for symmetric pairs are analogous to each other, therefore we omit those of t_2 with either t_3 or t_4. For the combination t_1 and t_2 we only show assertions of t_1 with statements of t_2.

Assertions in $\mathbf{t_1}$ $\{true\}$ is interference-free with all other instructions by Stable-ld or Stable-wr, depending on the statement. Statements 2 and 3 are the only ones changing $\mathtt{a}$ or $\mathtt{x}$, therefore all other statements do not interfere with $\{a \neq 1 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$ by Stable-ld or Stable-wr.

Statement 2

(I) by Stable-wr

$\{\mathtt{a} \neq 1\}\mathtt{x} := 2\{\mathtt{a} \neq 1\}$

(II) by Wr-1wr-single

$\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{0}\}\mathtt{x} := 2\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$

by (I), (II), Conj and Cons

$\{\mathtt{c} \neq 2 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{01} \land \mathtt{a} \neq 1 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$

$\{\mathtt{a} \neq 1 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{0}\}$

$\mathtt{x} := 2$

$\{\mathtt{a} \neq 1 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$

Statement 3

(I) by Ld-single

$\{t_0 \ltimes [\mathtt{x} \neq 1]\}\mathtt{a} \leftarrow \mathtt{x}\{\mathtt{a} \neq 1\}$

(II) by Stable-ld

$\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}\mathtt{a} \leftarrow \mathtt{x}\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$

by (I), (II), Conj and Cons

$\{(t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \vee \mathbb{1}^{\mathtt{x}}_{021}) \wedge \mathtt{a} \neq 1 \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$
$\{t_0 \ltimes [\mathtt{x} \neq 1] \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$
$\mathtt{a} \leftarrow \mathtt{x}$
$\{\mathtt{a} \neq 1 \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$

Assertions in $\mathbf{t_2}$ These are stable by reasoning analogous to t_1.

Assertions in $\mathbf{t_3}$ Statements of t_4 do not interfere with the assertions by Stable-ld, because neither `c` nor `d` are used in the assertions. Next we'll show that statement 1 from t_1 does not interfere with the assertions:

(I) by Wr-1wr-single and Cons

$\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\}$
$\{t_0 \ltimes \mathbb{1}_{\mathtt{x}} \wedge [x = 2]\}$
$\mathtt{x} := 1$
$\{t_0 \ltimes (\mathbb{1}_{\mathtt{x}} \wedge [x = 2]); (\mathbb{1}_{\mathtt{x}} \wedge [x = 1])\}$
$\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{21}\}$

pre(3) by (I), Wr-1wr and Cons

$\{\mathtt{a} \neq 1 \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02} \wedge t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \vee \mathbb{1}^{\mathtt{x}}_{021}\}$
$\{t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02}\}$
$\{t_0 \ltimes (\mathbb{1}_{\mathtt{x}} \wedge [x = 0]); \mathbb{1}^{\mathtt{x}}_{2}\}$
$\mathtt{x} := 1$

$$\left\{ t_0 \ltimes \begin{pmatrix} ((\mathbb{1}_{\mathtt{x}} \wedge [x = 0]); \mathbb{1}^{\mathtt{x}}_{1}; \mathbb{1}^{\mathtt{x}}_{2}) \vee \\ ((\mathbb{1}_{\mathtt{x}} \wedge [x = 0]); \mathbb{1}^{\mathtt{x}}_{21}) \end{pmatrix} \right\}$$

$\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{012} \vee \mathbb{1}^{\mathtt{x}}_{021}\}$

pre(4) by Stable-wr and Cons

$$\left\{ \begin{matrix} \mathtt{a} \neq 1 \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02} \wedge \\ (\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}) \end{matrix} \right\}$$

$\{\mathtt{a} \neq 1\}$
$1 : \mathtt{x} := 1$
$\{\mathtt{a} \neq 1\}$
$\{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}$

post(4) by Stable-wr and Cons

$$\left\{ \begin{matrix} \mathtt{a} \neq 1 \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{02} \wedge \\ ((\mathtt{a} = 1 \wedge \mathtt{b} = 2) \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}) \end{matrix} \right\}$$

$\{\mathtt{a} \neq 1\}$
$1 : \mathtt{x} := 1$
$\{\mathtt{a} \neq 1\}$
$\{(\mathtt{a} = 1 \wedge \mathtt{b} = 2) \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\}$

Finally we need to show that statement 2 from t_2 does not interfere with the assertions. Noninterference with pre(3) holds by the same reasoning as with statement 1.

(I) by Stable-wr

$\{\mathtt{a} \neq 1\}\mathtt{x} := 2\{\mathtt{a} \neq 1\}$

(II) by Wr-1wr-single

$\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{1}\}\mathtt{x} := 2\{t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}$

pre(4) by (I), (II), Disj2 and Cons

$\{\mathtt{c} \neq 2 \wedge t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{01} \wedge (\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12})\}$
$\{\mathtt{a} \neq 1 \vee t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{1}\}$
$\mathtt{x} := 1$
$\{\mathtt{a} \neq 1 \vee t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}$
$\{\mathtt{a} = 1 \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{12}\}$

post(4) by Stable-wr and Cons

$$\{\mathtt{c} \neq 2 \land t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{01} \land ((\mathtt{a} = 1 \land \mathtt{b} = 2) \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2})\}$$
$$\{\mathtt{a} \neq 1 \lor \mathtt{b} \neq 2 \lor (t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{01} \land \mathbb{1}^{\mathtt{x}}_{2})\}$$
$$\{\mathtt{a} \neq 1 \lor \mathtt{b} \neq 2\}$$
$$\mathtt{x} := 1$$
$$\{\mathtt{a} \neq 1 \lor \mathtt{b} \neq 2\}$$
$$\{(\mathtt{a} = 1 \land \mathtt{b} = 2) \Rightarrow t_3 \ltimes \mathbb{1}^{\mathtt{x}}_{2}\}$$

Assertions in $\mathbf{t_4}$ Noninterference with statement 1 and 2 is analogous to that of the assertions of t_3 with statement 2 and 1, respectively. Statements of t_3 do not interfere with the assertions by Stable-ld, because neither `a` nor `b` are used in the assertions. □

D.4 Message Passing

In this section, we prove Theorem 5.4, the validity of the proof outline for message passing seen below.

$$\textbf{Init} : \mathtt{x} := 0\,;\, \mathtt{y} := 0;$$
$$\{t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_{0} \land \mathbb{1}^{\mathtt{y}}_{0})\}$$
$$\begin{array}{l||l}
\textbf{Thread } \mathbf{t_1} & \textbf{Thread } \mathbf{t_2} \\
\{t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_{0} \land \mathbb{1}^{\mathtt{y}}_{0})\} & \{t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\} \\
1 : \mathtt{x} := 1; & 3 : \mathtt{a} \leftarrow^{A} \mathtt{y}; \\
\{t_1 \ltimes ([\mathtt{x} = 1] \land \mathbb{1}_{\mathtt{y}}) \land t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\} & \{\mathtt{a} = 1 \Rightarrow t_2 \ltimes [\mathtt{x} = 1]\} \\
2 : \mathtt{y} :=^{R} 1 & 4 : \mathtt{b} \leftarrow \mathtt{x} \\
\{\mathit{true}\} & \{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}
\end{array}$$
$$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$$

Proof. **Thread $\mathbf{t_1}$**

1 by Wr-own-1wr, Stable-wr, Conj and Cons

$$\{t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_{0} \land \mathbb{1}^{\mathtt{y}}_{0})\}$$
$$\{t_1 \ltimes \mathbb{1}_{\mathtt{x}} \land t_1 \ltimes \mathbb{1}_{\mathtt{y}} \land t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\}$$
$$\mathtt{x} := 1$$
$$\{t_1 \ltimes [\mathtt{x} = 1] \land t_1 \ltimes \mathbb{1}_{\mathtt{y}} \land t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\}$$
$$\{t_1 \ltimes ([\mathtt{x} = 1] \land \mathbb{1}_{\mathtt{y}}) \land t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\}$$

2 by Stable-wr and Cons

$$\{t_1 \ltimes ([\mathtt{x} = 1] \land \mathbb{1}_{\mathtt{y}}) \land t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\}\{\mathit{true}\}\mathtt{y} :=^{R} 1\{\mathit{true}\}$$

Thread $\mathbf{t_2}$

(I) by Ld-a-shift and Stable-ld

$\{t_2 \ltimes [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\}$

$\mathtt{a} \leftarrow^{A} \mathtt{y}$

$\{t_2 \ltimes [\mathtt{x} = 1]\}$

3 by (I), Ld-shift and Cons

$\{t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\}$

$\mathtt{a} \leftarrow^{A} \mathtt{y}$

$\{\mathtt{a} = 0 \vee t_2 \ltimes [\mathtt{x} = 1]\}$

$\{\mathtt{a} = 1 \implies t_2 \ltimes [\mathtt{x} = 1]\}$

4 by Stable-ld, Ld-single and Disj2

$\{\mathtt{a} = 1 \implies t_2 \ltimes [\mathtt{x} = 1]\}$

$\{\mathtt{a} \neq 1 \vee t_2 \ltimes [\mathtt{x} = 1]\}$

$4 : \mathtt{b} \leftarrow \mathtt{x}$

$\{\mathtt{a} \neq 1 \vee \mathtt{b} = 1\}$

$\{\mathtt{a} = 1 \implies \mathtt{b} = 1\}$

Noninterference

Statements from thread t_2 do not interfere with assertions in t_1 by Stable-ld, because the assertions neither contain a nor b. Similarly Statements 1 and 2 do not interfere with *post*(4) by Stable-wr, because the assertion does not contain either x or y.

statement 1 and pre(3) by Stable-wr and Cons

$\{t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_{0} \wedge \mathbb{1}^{\mathtt{y}}_{0}) \wedge t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\}$

$\{t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\}$

$\mathtt{x} := 1$

$\{t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\}$

$\{t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\}$

statement 1 and pre(4) by Stable-wr and Cons

$\{t_0 \ltimes (\mathbb{1}^{\mathtt{x}}_{0} \wedge \mathbb{1}^{\mathtt{y}}_{0}) \wedge (\mathtt{a} = 1 \implies t_2 \ltimes [\mathtt{x} = 1])\}$

$\{\mathtt{a} \neq 1 \vee (t_0 \ltimes \mathbb{1}^{\mathtt{x}}_{0} \wedge t_2 \ltimes [\mathtt{x} = 1])\}$

$\{\mathtt{a} \neq 1\}$

$\mathtt{x} := 1$

$\{\mathtt{a} \neq 1\}$

$\{\mathtt{a} = 1 \implies t_2 \ltimes [\mathtt{x} = 1]\}$

statement 2 and pre(3) by Wr-r-top and Cons

$\{t_1 \ltimes ([\mathtt{x} = 1] \wedge \mathbb{1}_{\mathtt{y}}) \wedge t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0} \wedge t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\}$

$\{t_1 \ltimes ([\mathtt{x} = 1] \wedge \mathbb{1}_{\mathtt{y}}) \wedge t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}\}$

$\mathtt{y} :=^{R} 1$

$\{t_2 \ltimes \mathbb{1}^{\mathtt{y}}_{0}; [\overline{A}(\mathtt{y})]; [\mathtt{x} = 1]\}$

```
statement 2 and pre(4) by Stable-wr and Cons
{$t_1 \ltimes ([\mathtt{x}=1] \wedge \mathbb{1}_{\mathtt{y}}) \wedge t_2 \ltimes \mathbb{1}^{\mathtt{y}}_0 \wedge (\mathtt{a}=1 \implies t_2 \ltimes [\mathtt{x}=1])$}
{$\mathtt{a}=1 \implies t_2 \ltimes [\mathtt{x}=1]$}
$\mathtt{y} :=^R 1$
{$\mathtt{a}=1 \implies t_2 \ltimes [\mathtt{x}=1]$}
```

□

D.5 Peterson's Algorithm

In this section, we show the mutual exclusion in the proof outline for Peterson's algorithm (as in [DDDW20]) seen below and show its validity, i. e. prove Theorem 5.5. We only displays thread t_1, thread t_2 is symmetric. $\mathtt{fl}_i, \mathtt{tu}_i, \mathtt{a}_i \in Var_L$ are local variables and $\mathtt{flag}_i, \mathtt{turn} \in Var_G$ are global variables for $i \in \{0,1\}$. We introduce the shortcut $C^x_e \equiv t_0 \ltimes C_x; \mathbb{1}^x_e$.

```
Init : $\mathtt{flag}_1 := false$ ; $\mathtt{flag}_2 := false$ ; $\mathtt{turn} := 0$ ; $\mathtt{a}_1 := false$ ; $\mathtt{a}_2 := false$;
 Thread t1
 {$\neg \mathtt{a}_1 \wedge t_1 \ltimes \mathbb{1}_{\mathtt{flag}_1} \wedge C^{\mathtt{turn}}_{\mathtt{turn}} \wedge (\neg \mathtt{a}_2 \vee (C^{\mathtt{turn}}_1 \wedge t_1 \ltimes C_{\mathtt{turn}}; \overline{A}(\mathtt{turn}); [\mathtt{flag}_2]))$}
1: $\mathtt{flag}_1 := true$;
 {$\neg \mathtt{a}_1 \wedge t_1 \ltimes [\mathtt{flag}_1] \wedge C^{\mathtt{turn}}_{\mathtt{turn}} \wedge (\neg \mathtt{a}_2 \vee (C^{\mathtt{turn}}_1 \wedge t_1 \ltimes C_{\mathtt{turn}}; \overline{A}(\mathtt{turn}); [\mathtt{flag}_2]))$}
2: $\langle \mathtt{turn.swap}(2)^{\mathrm{RA}}$ ; $\mathtt{a}_1 := true \rangle$
 {$\mathtt{a}_1 \wedge (\neg \mathtt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1 \vee t_1 \ltimes [\mathtt{flag}_2 \wedge \mathtt{turn} \neq 1])$}
 do
3:   $\mathtt{fl}_1 \leftarrow^A \mathtt{flag}_2$;
     {$\mathtt{a}_1 \wedge (\neg \mathtt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1 \vee (t_1 \ltimes [\mathtt{flag}_2 \wedge \mathtt{turn} \neq 1] \wedge \mathtt{fl}_1))$}
4:   $\mathtt{tu}_1 \leftarrow \mathtt{turn}$
     {$\mathtt{a}_1 \wedge (\neg \mathtt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1 \vee (t_1 \ltimes [\mathtt{flag}_2 \wedge \mathtt{turn} \neq 1] \wedge \mathtt{fl}_1 \wedge \mathtt{tu}_1 \neq 1))$}
5: until $(\neg \mathtt{fl}_1 \vee \mathtt{tu}_1 = 1)$ do
 {$\mathtt{a}_1 \wedge (\neg \mathtt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1)$}
6: Critical section;
 {$true$}
7: $\langle \mathtt{flag}_1 :=^R false$ ; $\mathtt{a}_1 := false \rangle$
 {$true$}
```

This example is special in that we are not interested in the postcondition, but in that no two threads can be in the critical section at the same time. We can show this by contradiction, assuming that a thread would be. Then it needed to fulfill *pre*(6) for both threads: $\mathtt{a}_1 \wedge (\neg \mathtt{a}_2 \vee t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1)$ and $\mathtt{a}_2 \wedge (\neg \mathtt{a}_1 \vee t_1 \ltimes \mathbb{1}^{\mathtt{turn}}_2)$. From their conjunction we can derive $t_2 \ltimes \mathbb{1}^{\mathtt{turn}}_1 \wedge t_1 \ltimes \mathbb{1}^{\mathtt{turn}}_2$, which is false by Theorem 3.13. Thus, our assumption must be false and no such state can exist.

Next, we prove Theorem 5.5, the validity of the proof outline.

Proof. Since the threads are symmetric, it is sufficient to only show the correctness for thread t_1 and the noninterference of the statements of t_2 with the assertions of t_1.

Because the correctness proof is rather large, we only show the parts which cannot be derived with STABLE-LD or STABLE-WR. These can be combined with the parts we show and the rules CONJ and DISJ1 to show the Hoare triples with the full assertions from the proof outline.

Thread $\mathbf{t_1}$

statement 1 by WR-OWN-1WR

$$\{t_1 \ltimes \mathbb{1}_{\texttt{flag}}\}\ \texttt{flag}_1 := \mathit{true}\ \{t_1 \ltimes \mathbb{1}^{\texttt{flag}}_{\mathit{true}}\}\{t_1 \ltimes [\texttt{flag}]\}$$

statement 2 by SUBST-ASGN

$$\{\mathit{true}\}\ \texttt{a}_1 := \mathit{true}\ \{\texttt{a}_1\}$$

statement 2

$$\begin{array}{l}
\{C^{\texttt{turn}}_{\texttt{turn}}\} \\
\{t_1 \ltimes C_{\texttt{turn}}; \mathbb{1}_{\texttt{turn}}\}\textsc{Cons} \\
\{t_1 \ltimes \mathbb{1}_{\texttt{turn}}\}\textsc{Wr-cvd} \\
\texttt{turn}.\mathbf{swap}(2)^{\mathrm{RA}} \\
\{t_1 \ltimes \mathbb{1}^{\texttt{turn}}_{2}\}\textsc{Wr-own-1wr} \\
\{t_1 \ltimes [\texttt{turn} \neq 1]\}\textsc{Cons}
\end{array}$$

statement 2

$$\begin{array}{l}
\{t_1 \ltimes C_{\texttt{turn}}; \overline{A}(\texttt{turn}); [\texttt{flag}_2]\} \\
\{t_1 \ltimes \overline{A}(\texttt{turn}); [\texttt{flag}_2]\}\textsc{Wr-cvd} \\
\{t_1 \ltimes [\texttt{flag}_2]\}\textsc{Swap-a-shift} \\
\texttt{turn}.\mathbf{swap}(2)^{\mathrm{RA}} \\
\{t_1 \ltimes [\texttt{flag}_2]\}\textsc{Stable-wr}
\end{array}$$

statement 3 by LD-SINGLE

$$\{t_1 \ltimes [\texttt{flag}_2]\}\ \mathit{fl}_1 \leftarrow^{A} \texttt{flag}_2\ \{\texttt{fl}_1\}$$

statement 4 by LD-SINGLE

$$\{t_1 \ltimes [\texttt{turn} \neq 1]\}\ \mathit{fl}_1 \leftarrow^{A} \texttt{turn}\ \{\texttt{tu}_1 \neq 1\}$$

statement 5 by WHILE and CONS

$$\begin{array}{l}
\texttt{a}_1 \land \begin{pmatrix} \neg\texttt{a}_2 \lor t_2 \ltimes \mathbb{1}^{\texttt{turn}}_{1} \lor \\ (t_1 \ltimes [\texttt{flag}_2 \neq 0 \land \texttt{turn} \neq 1] \land \texttt{fl}_1 \land \texttt{tu}_1 \neq 1) \end{pmatrix} \land (\texttt{fl}_1 = 0 \lor \texttt{tu}_1 = 1) \\
\implies \texttt{a}_1 \land (\neg\texttt{a}_2 \lor t_2 \ltimes \mathbb{1}^{\texttt{turn}}_{1})
\end{array}$$

Noninterference

Because the threads are symmetric, we only show the noninterference of the statements of t_2 with the assertions of t_1.

Statement 3: $\mathit{fl}_2 \leftarrow^{A} \texttt{flag}_1$ and 4: $\mathit{tu}_2 \leftarrow \texttt{turn}$ preserve all assertions by STABLE-LD. Statement 1: $\texttt{flag}_2 := 1$ and 7: $\langle \texttt{flag}_2 :=^{R} 0\,; \texttt{a}_2 := \mathit{false}\rangle$ preserve all parts of assertions by STABLE-WR, except assertions of the form $\neg\texttt{a}_2 \lor \varphi(\texttt{flag}_2)$. Those assertions hold because we can guarantee $\neg\texttt{a}_2$ after the statements are executed by STABLE-WR for statement 1 since $\neg\texttt{a}_2$ is in its precondition and SUBST-ASGN for statement 7.

The truly interesting statement is 2: $\langle \texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}\,; \texttt{a}_2 := \mathit{true}\rangle$. There are four assertions this needs to preserve: $C^{\texttt{turn}}_{1}$, $C^{\texttt{turn}}_{\texttt{turn}}$, $C^{\texttt{turn}}_{1} \land t_1 \ltimes C_t; \overline{A}(\texttt{turn}); \mathbb{1}^{\texttt{flag}_2}_{1}$ and $t_2 \ltimes \mathbb{1}^{\texttt{turn}}_{1}$. We show these for $\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$, this can be combined with $\texttt{a}_2 := \mathit{true}$ which preserves all of the postconditions by SUBST-ASGN. We begin with the last which is the simplest.

$$t_2 \ltimes \mathbb{1}_1^{\texttt{turn}} \text{ by } \textsc{Wr-own-1wr}$$
$$\{C \ltimes \mathbb{1}_{\texttt{turn}}^{\texttt{turn}}\}\{t_2 \ltimes C\texttt{turn}; \mathbb{1}_{\texttt{turn}}\}\ \texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}\ \{t_2 \ltimes \mathbb{1}_1^{\texttt{turn}}\}$$

For $C_1^{\texttt{turn}}$ we need to derive $t_i \ltimes C_{\texttt{turn}}; \mathbb{1}_1^{\texttt{turn}}$ for every thread t_i. Then we can derive $\{C_{\texttt{turn}}^{\texttt{turn}}\}$ $\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$ $\{C_1^{\texttt{turn}}\}$ by applying Conj for all threads.

$$t_i \ltimes C_1^{\texttt{turn}} \text{ by } \textsc{Swap-wr}$$
$$\{C_{\texttt{turn}}^{\texttt{turn}}\}\{t_2 \ltimes C_{\texttt{turn}}; \mathbb{1}_{\texttt{turn}} \wedge t_i \ltimes C_{\texttt{turn}}; \mathbb{1}_{\texttt{turn}}\}\ \texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}\ \{t_i \ltimes C_{\texttt{turn}}; \mathbb{1}_1^{\texttt{turn}}\}$$

Both $C_1^{\texttt{turn}}$ and $C_{\texttt{turn}}^{\texttt{turn}}$ then are stable by Cons.

For stability of $C_1^{\texttt{turn}} \wedge t_1 \ltimes C_t; \overline{A}(\texttt{turn}); [\texttt{flag}_2]$ we reuse $\{C_{\texttt{turn}}^{\texttt{turn}}\}$ $\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$ $\{C_{\texttt{turn}}^{\texttt{turn}}\}$. Additionally, we need $t_2 \ltimes [\texttt{flag}_2]$ from the statements precondition.

(I), see above

$\{C_1^{\texttt{turn}}\}$
$\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$
$\{C_1^{\texttt{turn}}\}$

(II) by Swap-r

$\{C_1^{\texttt{turn}} \wedge t_2 \ltimes [\texttt{flag}_2]\}$
$\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$
$\{t_1 \ltimes C_t; \overline{A}(\texttt{turn}); [\texttt{flag}_2]\}$

$C_1^{\texttt{turn}} \wedge t_1 \ltimes C_t; \overline{A}(\texttt{turn}); \mathbb{1}_1^{\texttt{flag}_2}$ by (I), (II), Cons and Conj

$\{C_{\texttt{turn}}^{\texttt{turn}} \wedge t_1 \ltimes C_t; \overline{A}(\texttt{turn}); [\texttt{flag}_2] \wedge t_2 \ltimes [\texttt{flag}_2]\}$
$\{C_{\texttt{turn}}^{\texttt{turn}} \wedge C_1^{\texttt{turn}} \wedge t_2 \ltimes [\texttt{flag}_2]\}$
$\texttt{turn}.\mathbf{swap}(1)^{\mathrm{RA}}$
$\{C_1^{\texttt{turn}} \wedge t_1 \ltimes C_t; \overline{A}(\texttt{turn}); [\texttt{flag}_2]\}$

□