

Formal Definition and Safety Proof for the SPLASH 2010 Submission

Nicholas D. Matsakis

ETH Zurich

Abstract. This document contains a complete formal definition and safety proof for the Intervals Type System contained in our submission for SPLASH 2010.

1 Introduction

This document contains a complete formal definition and safety proof for Inter, the language and type system contained in our submission for SPLASH 2010. We assume familiarity with the basic structure of the type system and language. We will prove that any Inter program making exclusive use of Interval and Lock guards is data-race free.

We begin by briefly introducing the grammar as well as the operational semantics. We then cover the typing rules from SPLASH 2010. Finally, we introduce the safety theorem and its associated lemmas.

2 Operational Semantics

The grammar of Inter is given in Figure 1. The state of the Inter machine is described as a tuple $(\mathcal{E}; \mathcal{O}; \mathcal{A})$.

- The environment \mathcal{E} contains a set of tuples which define the types of all local variables, values of all fields, and scheduling constraints. These tuples take the form:
 - $x : ty$: local variable x has type ty .
 - $x \text{ hb } x'$: interval x *happens before* x' .
 - $x \text{ subOf } x'$: interval x is an asynchronous subinterval of x' .
 - $x \text{ inlineSubOf } x'$: interval x is an inline subinterval of x' .
 - $x \text{ locks } x'$: interval x will hold the lock x' .
 - $x.f \text{ eq } x'$: the value of the field f of variable x is equal to x' .

The tuples in the environment are restricted to those that are directly created by statements in the grammar.

Note that this environment is a subset of the environment used in type check. The environment used in the type check can include all the tuples shown above, but may additionally include tuples of the form *path rel path* which relate arbitrary paths via arbitrary relations.

```

cdecl  := class c(tys fs) extends c(paths) { members }
member := gdecl | fdecl | mdecl | idecl | ldecl | hbdecl
gdecl  := ghost f
fdecl  := ty f guardedBy path
mdecl  := void m(tys xs) reqs { lstmts }
req    := path rel path
idecl  := interval f(path) { lstmts }
ldecl  := path locks path
hbdecl := path hb path
path   := x.fs
rel    := trel | wcrl | locks
trel   := sub0f | inlineSub0f | hb
wcrl   := permitsWr | permitsRd | ensuresImm | eq
lstmt  := x : stmt
stmt   := x.f = x
        | x = x.f
        | x = new c[fs eq paths] (xs)
        | x.m(xs)
        | assert x wcrl x
ty     := c[fs wcrls paths]

```

Fig. 1. Grammar for Inter

- \mathcal{O} contains a set of points which have occurred. A point has the form $x.f$ where f is either `start`, `mid`, or `end`.
 - When $x.start$ has occurred, the interval x has commenced execution.
 - When $x.mid$ has occurred, the interval’s code block is complete and so asynchronous subintervals may commence execution.
 - When $x.end$ has occurred, the interval x has completed execution in its entirety.
- $\mathcal{A} : x \rightarrow lstmts$ maps an asynchronous interval x to its statements $lstmts$.

Null Pointers We handle null pointers in the following way. There is assumed to be an infinite set `null` of unique variable names corresponding to null pointers. No variable in this set may be legally dereferenced. The initial value of most fields is a fresh value from this set. All values of this set are equivalent to the operational semantics.

The need for the set `null` is an artifact of the small step semantics formalization. We will see that when a variable’s value is determined, that variable is replaced in later steps with its value. If we had only a single value for `null`, then all variables to which `null` were assigned would be equivalent. This is not the case in the language itself, however: there, a variable cannot change its type, and so two variables which are both assigned `null` still have distinct types.

High-Level Machine Steps The various steps that the Inter machine can take are given in Figure 2. Each takes the form $(\mathcal{E}; \mathcal{O}; \mathcal{A}) \rightarrow (\mathcal{E}'; \mathcal{O}'; \mathcal{A}')$.

$$\begin{array}{c}
\text{STEP-START} \\
\frac{\mathcal{E}; \mathcal{O} \vdash \mathbf{x} \text{ ready}}{(\mathcal{E}; \mathcal{O}; \mathcal{A}) \rightarrow (\mathcal{E}; \mathcal{O} + \mathbf{x.start}; \mathcal{A})} \\
\\
\text{STEP-INLINE} \\
\frac{\mathcal{A}(\mathbf{x}) = lstmts = (\mathbf{x}_l : \dots) + \dots \quad \mathbf{x}_l.start \in \mathcal{O} \quad (\mathcal{E}; lstmts) \rightarrow (\mathcal{E}'; lstmts') + \mathcal{A}_n}{(\mathcal{E}; \mathcal{O}; \mathcal{A}) \rightarrow (\mathcal{E}'; \mathcal{O}; \mathcal{A} + \mathcal{A}_n + (\mathbf{x} \rightarrow lstmts'))} \\
\\
\text{STEP-MID} \\
\frac{\mathbf{x.start} \in \mathcal{O} \quad \nexists \mathbf{x}_i. \mathcal{A}(\mathbf{x}_i) = (\mathbf{x} : \dots) + \dots \quad \forall \mathbf{x}_c. (\mathbf{x}_c \text{ inlineSubOf } \mathbf{x} \in \mathcal{E}) \Rightarrow (\mathbf{x}_c.end \in \mathcal{O})}{(\mathcal{E}; \mathcal{O}; \mathcal{A}) \rightarrow (\mathcal{E}; \mathcal{O} + \mathbf{x.mid}; \mathcal{A})} \\
\\
\text{STEP-END} \\
\frac{\mathbf{x.mid} \in \mathcal{O} \quad \forall \mathbf{x}_c. (\mathbf{x}_c \text{ subOf } \mathbf{x} \in \mathcal{E}) \Rightarrow (\mathbf{x}_c.end \in \mathcal{O})}{(\mathcal{E}; \mathcal{O}; \mathcal{A}) \rightarrow (\mathcal{E}; \mathcal{O} + \mathbf{x.end}; \mathcal{A})}
\end{array}$$

Fig. 2. Steps of the Inter machine.

- STEP-START starts the interval \mathbf{x} . \mathbf{x} must be **ready** to execute. The rules for readiness are given shortly. The summary is that all predecessors must have completed and all locks to be acquired must be available.
- STEP-INLINE executes the next statement from the asynchronous interval \mathbf{x} . The statement is also the body for the inline subinterval \mathbf{x}_l , which must have started already. After executing the statement:
 - \mathcal{E}' contains the new environment, which may have updated fields, new variables, or updated relations.
 - $lstmts'$ represents the remaining statements for that \mathbf{x} . These may have been substituted or otherwise transformed.
 - \mathcal{A}_n represents new asynchronous intervals created by the statement.
Note that in the new environment the inline subinterval \mathbf{x}_l is not yet considered to be complete, only to have reached its mid point. This is because \mathbf{x}_l may still have asynchronous subintervals that have not yet executed.
- STEP-MID allows an interval \mathbf{x} to reach its mid point once its code has executed and all of its inline subintervals have completed. This rule can be used for both inline and asynchronous intervals. The second clause, $\nexists \mathbf{x}_i$, only applies to inline intervals (for asynchronous intervals, it is always satisfied).
- STEP-END allows an interval \mathbf{x} to reach its end point once
 1. its mid point has occurred; and,
 2. all of its asynchronous subintervals have completed.

Readiness The rules for readiness are given Figure 3. An interval \mathbf{x} is **ready** when its start point could legally occur.

$$\begin{array}{c}
\text{READY-ASYNC} \\
\frac{\begin{array}{c}
\mathbf{x} \text{ subOf } \mathbf{x}_p \in \mathcal{E} \quad \mathbf{x}_p.\text{mid} \in \mathcal{O} \\
\forall \mathbf{x}_b. (\mathbf{x}_b \text{ hb } \mathbf{x} \in \mathcal{E}) \Rightarrow (\mathbf{x}_b.\text{end} \in \mathcal{O}) \\
\forall \mathbf{x}_l, \mathbf{x}_i. (\mathbf{x} \text{ locks } \mathbf{x}_l \in \mathcal{E} \wedge \mathbf{x}_i \text{ locks } \mathbf{x}_l \in \mathcal{E}) \Rightarrow (\mathbf{x}_i.\text{end} \in \mathcal{O} \vee \mathbf{x}_i.\text{start} \notin \mathcal{O})
\end{array}}{\mathcal{E}; \mathcal{O} \vdash \mathbf{x} \text{ ready}} \\
\\
\text{READY-INLINE} \\
\frac{\begin{array}{c}
\mathbf{x} \text{ inlineSubOf } \mathbf{x}_p \in \mathcal{E} \quad \mathbf{x}_p.\text{start} \in \mathcal{O} \\
\forall \mathbf{x}_b. (\mathbf{x}_b \text{ hb } \mathbf{x} \in \mathcal{E}) \Rightarrow (\mathbf{x}_b.\text{end} \in \mathcal{O}) \\
\forall \mathbf{x}_l, \mathbf{x}_i. (\mathbf{x} \text{ locks } \mathbf{x}_l \in \mathcal{E} \wedge \mathbf{x}_i \text{ locks } \mathbf{x}_l \in \mathcal{E}) \Rightarrow (\mathbf{x}_i.\text{end} \in \mathcal{O} \vee \mathbf{x}_i.\text{start} \notin \mathcal{O})
\end{array}}{\mathcal{E}; \mathcal{O} \vdash \mathbf{x} \text{ ready}}
\end{array}$$

Fig. 3. Readiness.

- READY-ASYNC states that an asynchronous interval \mathbf{x} is ready if
 - the parent interval \mathbf{x}_p has reached its mid point;
 - any intervals which *happen before* \mathbf{x} have ended;
 - no active interval holds a lock which \mathbf{x} requires.
- READY-INLINE gives the rules for an inline interval \mathbf{x} . They are the same as the asynchronous case except that the parent interval \mathbf{x}_p need only have started, not reach its mid point.

Executing Statements Figure 4 gives the rules for executing a single statement.

- EXEC-STORE handles stores. The rule simply updates the environment with a new eq relation.
- EXEC-LOAD handles loads. The environment must contain a tuple $(\mathbf{x}_o.f \text{ eq } \mathbf{x}_v)$ giving the value \mathbf{x}_v for the field. The result variable \mathbf{x}_d is replaced with \mathbf{x}_v .
- EXEC-NEW handles new statements. The rule creates a fresh variable \mathbf{x}_n that must not exist in the program text or in the environment. It then uses a helper function *initial()* (defined in the next section) to enrich the environment with the initial values of the fields of \mathbf{x}_n . Finally, the helper function *asyncMembers()* returns the bodies for any interval members of \mathbf{x}_n .
- EXEC-METHOD handles method calls by replacing the call with an inlined version of the method being invoked. The rule involves a large number of variables. Each one is defined here:
 - \mathcal{E}_0 : the initial environment
 - \mathbf{x}_l : the label of the method call
 - \mathbf{x}_r : the receiver of the method
 - \mathbf{m} : the method name being called
 - \mathbf{xs}_a : the variables used for each argument
 - lstmts_r : the remaining statements after the method call

2.1 Executing Statements

$$\begin{array}{c}
\text{EXEC-STORE} \\
\frac{x_o \notin \text{null} \quad \mathcal{E}' = (\mathcal{E} \setminus x_o.f) + (x_o.f \text{ eq } x_v)}{(\mathcal{E}; (x_l : x_o.f = x_v) + lstmts_r) \rightarrow (\mathcal{E}'; lstmts_r) + []} \\
\\
\text{EXEC-LOAD} \\
\frac{x_o \notin \text{null} \quad (x_o.f \text{ eq } x_v) \in \mathcal{E}}{(\mathcal{E}; (x_l : x_d = x_o.f) + lstmts_r) \rightarrow (\mathcal{E}; [x_d \rightarrow x_v] lstmts_r) + []} \\
\\
\text{EXEC-NEW} \\
\frac{\begin{array}{c} x_n \text{ fresh} \\ \mathcal{E}' = \mathcal{E} + (x_n : c[\text{fs eq paths}] + \text{initial}(\mathcal{E}, x_l, x_n, c[\text{fs eq paths}], xs)) \\ lstmts'_r = [x_d \rightarrow x_n](lstmts_r) \quad \mathcal{A}_n = \text{asyncMembers}(\mathcal{E}', x_n) \end{array}}{(\mathcal{E}; (x_l : x_d = \text{new } c[\text{fs eq paths}](xs)) + lstmts_r) \rightarrow (\mathcal{E}'; lstmts'_r) + \mathcal{A}_n} \\
\\
\text{EXEC-METHOD} \\
\frac{\begin{array}{c} x_r \notin \text{null} \quad (x_r : c[...]) \in \mathcal{E}_0 \\ mdecl(c, m) = \text{void } m(tys_m \ xs_m) \dots \{ lstmts_m \} \quad lstmts_m = xs_l : \dots \\ xs_n \text{ fresh} \quad \mathcal{E}_1 = \mathcal{E}_0 + (xs_n(i) : \text{Interval} \mid i) \\ \mathcal{E}_2 = \mathcal{E}_1 + (xs_n(i) \text{ inlineSubOf } x_l \mid i) \\ \mathcal{E}_3 = \mathcal{E}_2 + (xs_n(i-1) \text{ hb } xs_n(i) \mid i) \\ \Theta = [\text{this} \rightarrow x_r, \text{method} \rightarrow x_l, xs_m \rightarrow xs_a, xs_l \rightarrow xs_n] \end{array}}{(\mathcal{E}_0; (x_l : x_r.m(xs_a)) + lstmts_r) \rightarrow \mathcal{E}_3; \Theta(lstmts_m) + lstmts_r} \\
\\
\text{EXEC-ASSERT} \\
\frac{\mathcal{E} \vdash x_g \text{ wcrel } x_i}{(\mathcal{E}; (x_l : \text{assert } x_g \text{ wcrel } x_i) + lstmts_r) \rightarrow \mathcal{E}; lstmts_r}
\end{array}$$

Fig. 4. Executing individual statements.

- c : the type of the receiver
 - $tys_m \ xs_m$: the types and names of the methods parameters
 - $lstmts_m$: the statements in the body of the method being invoked
 - xs_l : the labels of $lstmts_m$
 - xs_n : fresh interval variables for each of $lstmts_m$
 - \mathcal{E}_1 : an environment containing declarations for each of xs_n
 - \mathcal{E}_2 : an environment containing `inlineSubOf` relations for each of xs_n
 - \mathcal{E}_3 : an environment containing `hb` relations between subsequent intervals in xs_n
- EXEC-ASSERT allows execution to proceed only if the given relation is provable in the environment \mathcal{E} .

The function *initial()* The function *initial()* is used in EXEC-NEW to enrich the environment with the initial values of the fields of x_n . The function is not defined in all cases; it can fail due to `null` pointers errors or other assertion

errors. The tuples added by $initial(\mathcal{E}, \mathbf{x}_l, \mathbf{x}_n, \mathbf{c}[\mathbf{fs} \text{ eq } \mathbf{paths}], \mathbf{xs})$ are defined as follows:

1. A tuple $(\mathbf{x}_n : \mathbf{c}[\mathbf{fs} \text{ eq } \mathbf{paths}])$ reflects the type of the new object.
2. Tuples $(\mathbf{x}_n.\mathbf{fs}(i) \text{ eq } \mathbf{xs}(i))$ for each of the constructor arguments \mathbf{fs} defined in \mathbf{c} . For each superclass of \mathbf{c} , the superclass arguments are derived by evaluating the paths from the class declaration. Similar tuples are added with the resulting values. During this evaluating, a temporary environment is used in which all non-constructor fields (including interval members) are initialized to a fresh null value.
3. For each reified field \mathbf{f} with type ty declared in \mathbf{c} or one of its superclasses, tuples $(\mathbf{x}_n.\mathbf{f} \text{ eq } \mathbf{x})$ and $(\mathbf{x} : ty)$ are added. Here $\mathbf{x} \in \text{null}$ represents a fresh null variable.
4. For each interval member \mathbf{f} with parent $path$ of \mathbf{c} or a superclass:
 - The parent $path$ is evaluated to \mathbf{x}_p .
 - Tuples $(\mathbf{x}_n.\mathbf{f} \text{ eq } \mathbf{x})$, $(\mathbf{x} : \text{Interval})$ and $(\mathbf{x} \text{ subOf } \mathbf{x}_p)$ are added, where \mathbf{x} is a fresh variable name.
5. For each *happens before* declaration $path_1 \text{ hb } path_2$, the two paths are evaluated to variables \mathbf{x}_1 and \mathbf{x}_2 . The result is undefined unless $\mathcal{E} \vdash \mathbf{x}_1 : \text{Interval}$ and $\mathcal{E} \vdash \mathbf{x}_2 : \text{Interval}$. Furthermore, \mathbf{x}_2 must either be an interval member of the object \mathbf{x}_n being created, or $\mathcal{E} \vdash \mathbf{x}_l \text{ hb } \mathbf{x}_2$. This ensures that \mathbf{x}_2 cannot yet have started. Assuming these conditions are met, a tuple $\mathbf{x}_1 \text{ hb } \mathbf{x}_2$ is added to the environment.
6. Lock declarations are handled analogously to *happens before* declarations. The only differences are that the variable \mathbf{x}_1 must be of lock type and that the final tuple to be added is $\mathbf{x}_1 \text{ locks } \mathbf{x}_2$.

In the definition of $initial()$, we made several references to “evaluating a path.” The rules for evaluating a path follow. Note that evaluating files is a null pointer would have to be dereferenced. If this result occurs, the function $initial()$ has an undefined result.

$$\frac{\text{EVAL-VAR} \quad \mathcal{E} \vdash \mathbf{x} \rightsquigarrow \mathbf{x} \quad \text{EVAL-PATH} \quad \mathcal{E} \vdash path \rightsquigarrow \mathbf{x}_p \quad \mathbf{x}_p \notin \text{null} \quad (\mathbf{x}_p.\mathbf{f} \text{ eq } \mathbf{x}_v) \in \mathcal{E}}{\mathcal{E} \vdash path.\mathbf{f} \rightsquigarrow \mathbf{x}_v}$$

3 Definitions

- Two accesses are *ordered* if the intervals in which they occur are ordered.
- Two intervals are *ordered* if any of the following conditions hold:
 - $\mathbf{x}_l = \mathbf{x}_r$
 - $\mathbf{x}_l \text{ hb } \mathbf{x}_r$
 - $\mathbf{x}_r \text{ hb } \mathbf{x}_l$
 - Both \mathbf{x}_l and \mathbf{x}_r acquire the same lock

$$\begin{array}{c}
\text{WF-STATE} \\
\frac{\mathcal{E} \vdash \mathcal{O} \quad \mathcal{E} \vdash \mathcal{A} \quad \forall i. \mathcal{E} \vdash \mathcal{E}(i)}{\vdash (\mathcal{E}; \mathcal{O}; \mathcal{A})} \\
\\
\text{WF-OCC} \\
\frac{\forall \mathbf{x}. \mathbf{f} \in \mathcal{O}. (\mathbf{x} : \text{Interval}) \in \mathcal{E} \\
(\mathbf{x}. \text{mid} \in \mathcal{O}) \Rightarrow (\mathbf{x}. \text{start} \in \mathcal{O}) \quad (\mathbf{x}. \text{end} \in \mathcal{O}) \Rightarrow (\mathbf{x}. \text{mid} \in \mathcal{O}) \\
\forall (\mathbf{x}_b \text{ hb } \mathbf{x}_a \in \mathcal{E}). (\mathbf{x}_a. \text{start} \in \mathcal{O}) \Rightarrow (\mathbf{x}_b. \text{end} \in \mathcal{O}) \\
\forall (\mathbf{x}_c \text{ inlineSubOf } \mathbf{x}_p \in \mathcal{E}). (\mathbf{x}_p. \text{mid} \in \mathcal{O}) \Rightarrow (\mathbf{x}_c. \text{end} \in \mathcal{O}) \\
\forall (\mathbf{x}_c \text{ subOf } \mathbf{x}_p \in \mathcal{E}). (\mathbf{x}_p. \text{end} \in \mathcal{O}) \Rightarrow (\mathbf{x}_c. \text{end} \in \mathcal{O})}{\mathcal{E} \vdash \mathcal{O}} \\
\\
\text{WF-ASYNC} \qquad \qquad \qquad \text{WF-ASYNC-EMPTY} \\
\frac{\forall \mathbf{x}. \mathcal{E} \vdash (\mathbf{x} \rightarrow \mathcal{A}(\mathbf{x}))}{\mathcal{E} \vdash \mathcal{A}} \qquad \qquad \frac{\mathbf{x} \notin \text{null} \quad (\mathbf{x} : \text{Interval}) \in \mathcal{E}}{\mathcal{E} \vdash (\mathbf{x} \rightarrow \square)} \\
\\
\text{WF-ASYNC-ENTRY} \\
\frac{\mathbf{x} \notin \text{null} \quad (\mathbf{x} : \text{Interval}) \in \mathcal{E} \\
BV(\text{lstm}ts) \# BV(\mathcal{E}) \quad FV(\text{lstm}ts) \subseteq BV(\mathcal{E}) \\
\text{lstm}ts = \mathbf{x}s_i : \dots \quad \forall i. (\mathbf{x}s_i(i) \text{ inlineSubOf } \mathbf{x}) \in \mathcal{E} \\
\text{stable}(\mathcal{E}, \mathbf{x}s_i(1)) \vdash \text{lstm}ts \xrightarrow{\text{fold}} \mathcal{E}'}{\mathcal{E} \vdash (\mathbf{x} \rightarrow \text{lstm}ts)}
\end{array}$$

Fig. 5. Well-formed machine states.

- \mathbf{x}_l (resp. \mathbf{x}_r) is a (possibly transitive) subinterval of \mathbf{x}_p and \mathbf{x}_r (resp. \mathbf{x}_l) is ordered w.r.t. \mathbf{x}_p

Note that ordering is a symmetric relation.

- An interval \mathbf{x} is *active* in a machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$ if $\mathbf{x}. \text{start} \in \mathcal{O}$ but $\mathbf{x}. \text{end} \notin \mathcal{O}$.
- The bound variables $BV(\mathcal{E})$ of an environment are those variables \mathbf{x} for which a typing relation $(\mathbf{x} : \text{ty})$ exists.
- The bound variables $BV(\text{lstm}ts)$ of a sequence of statements are those variables which are assigned to by a statement in the sequence. The free variables $FV(\text{lstm}ts)$ of a sequence of statements are those variables which are referenced before they are assigned. Note that the free and bound variables of all method and interval bodies are always disjoint as a basic well-formedness criteria of an Inter program (in other words, variables are never re-assigned, and are never used before they are assigned).
- \mathcal{E} is a sub-environment of \mathcal{E}' if $\mathcal{E} \vdash \mathbf{x} : \text{ty} \Rightarrow \mathcal{E}' \vdash \mathbf{x} : \text{ty}$ and $\mathcal{E} \vdash \text{path rel path}' \Rightarrow \mathcal{E}' \vdash \text{path rel path}'$.
- A machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$ is *well-formed* if the rule WF-STATE applies. The rule and related judgements are given in figures 5 and 6. They enforce a number of self-consistency conditions, such as that *happens before* edges

$$\begin{array}{c}
\text{WF-REL-VAR-DECL} \\
\frac{\mathcal{E} \vdash \mathbf{c}[\mathbf{fs} \text{ eq paths}] \quad \mathbf{x} \in \mathbf{null} \vee \text{class relations for } \mathbf{c} \text{ hold}}{\mathcal{E} \vdash \mathbf{x} : \mathbf{ty}} \\
\\
\text{WF-REL-HB} \\
\frac{\mathbf{x}_l, \mathbf{x}_r \notin \mathbf{null} \quad (\mathbf{x}_l : \mathbf{Interval}) \in \mathcal{E} \quad (\mathbf{x}_r : \mathbf{Interval}) \in \mathcal{E}}{\mathcal{E} \vdash (\mathbf{x}_l \text{ hb } \mathbf{x}_r)} \\
\\
\text{WF-REL-INLINE SUBOF} \\
\frac{\mathbf{x}_c, \mathbf{x}_p \notin \mathbf{null} \quad (\mathbf{x}_c : \mathbf{Interval}) \in \mathcal{E} \quad (\mathbf{x}_p : \mathbf{Interval}) \in \mathcal{E}}{\mathcal{E} \vdash (\mathbf{x}_c \text{ inlineSubOf } \mathbf{x}_p)} \\
\\
\text{WF-REL-SUBOF} \\
\frac{\mathbf{x}_c, \mathbf{x}_p \notin \mathbf{null} \quad (\mathbf{x}_c : \mathbf{Interval}) \in \mathcal{E} \quad (\mathbf{x}_p : \mathbf{Interval}) \in \mathcal{E}}{\mathcal{E} \vdash (\mathbf{x}_c \text{ subOf } \mathbf{x}_p)} \\
\\
\text{WF-REL-LOCKS} \\
\frac{\mathbf{x}_l, \mathbf{x}_r \notin \mathbf{null} \quad (\mathbf{x}_l : \mathbf{Interval}) \in \mathcal{E} \quad (\mathbf{x}_r : \mathbf{Lock}) \in \mathcal{E}}{\mathcal{E} \vdash (\mathbf{x}_l \text{ locks } \mathbf{x}_r)} \\
\\
\text{WF-REL-FIELD} \\
\frac{(\mathbf{x}_o : \mathbf{c}[\dots]) \in \mathcal{E} \quad \mathbf{x}_o \notin \mathbf{null} \quad \text{reified}(\mathbf{c}, \mathbf{f}) = (\mathbf{ty}; \dots) \quad \mathcal{E} \vdash \mathbf{x}_v : [\mathbf{this} \rightarrow \mathbf{x}_o]\mathbf{ty}}{\mathcal{E} \vdash (\mathbf{x}_o.\mathbf{f} \text{ eq } \mathbf{x}_v)}
\end{array}$$

Fig. 6. Well-formed relations.

only connect intervals, or that fields all have values of compatible type with their declaration.

The most interesting wrinkle is in WF-ASYNC-ENTRY, which determines when an asynchronous interval is well-formed. It states that the statements within an asynchronous interval must be typable in $\text{stable}(\mathcal{E}, \mathbf{x}_l)$. The function $\text{stable}()$ simply removes any $\mathbf{x}_o.\mathbf{f} \text{ eq } \mathbf{x}_v$ relation from \mathcal{E} unless it can be shown that $\mathcal{E} \vdash \mathbf{x}_o.\mathbf{f} \text{ stableBy } \mathbf{x}_l$. This rule expresses the notion that the typing of an interval cannot rely on eq relations that are not stable.

4 Properties of the Environment

We begin with several lemmas that establish various properties of a well-formed machine state.

Lemma 1 (Monotonic).

If \mathcal{E} is a sub-environment of \mathcal{E}' then $\mathcal{E} \vdash \text{lstmts} \xrightarrow{\text{fold}} \mathcal{E}_N \Rightarrow (\mathcal{E}' \vdash \text{lstmts} \xrightarrow{\text{fold}} \mathcal{E}'_N \wedge \mathcal{E}_N \subseteq \mathcal{E}'_N)$.

Proof. Immediate.

Lemma 2 (Uniqueness).

Given a well-formed machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$, for all paths $path$, there exists at most one variable x such that $\mathcal{E} \vdash path \text{ eq } x$.

Proof. Paths can be equated to a variable either through `eq` relations or through the type of some element of the path. We prove by induction that any list of equate steps which eventually leads to the variable x could never lead to another variable.

Because the environment is well-formed, the only `eq` relations which it contains are of the form $\mathcal{E} \vdash x_o.f \text{ eq } x_v$. Therefore, paths can only be shortened using an `eq` relation in a particular way, by replacing a prefix $x_o.f$ of the path with a single variable x_v . This will always lead to a single well-defined variable.

Lengthening a path $path$ using an `eq` relation leads to a path $path'$ that can only be shortened to $path$. Therefore, any number of lengthening steps can be undone in only one particular way, which always leads back to $path$.

Finally, we must consider ghost fields. For a ghost field, there is never an explicit `eq` relation in the environment. Therefore, a path $path.f$ ending in a ghost field f can only be equated by the type of the owner $path$. The owner can therefore relate $path.f$ to at most one other path $path'$. By induction $path'$ can only be shortened to x .

Lemma 3 (Path Type).

Given a well-formed machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$, a type derived via the rules T-VAR or T-FIELD is unique.

Proof. For T-VAR the lemma holds because all variables are only defined once in a well-formed environment.

For T-FIELD, the lemma holds because all field names are unique and declared in precisely one type. Therefore, for a given f , the function $reified(c, f)$ is either undefined or always yields the same result.

Lemma 4 (Variable Relations Only).

Given a non-wc-rel relation rel , environment \mathcal{E} , and paths $path_l, path_r$, such that:

- $\vdash \mathcal{E}$
- $\mathcal{E} \vdash path_l \text{ rel } path_r$

then $\exists x_l, x_r$ such that:

- $\mathcal{E} \vdash path_l \text{ eq } x_l$
- $\mathcal{E} \vdash path_r \text{ eq } x_r$
- $\mathcal{E} \vdash x_l \text{ rel } x_r$

Proof. By induction over the proof tree of $\mathcal{E} \vdash path_l \text{ rel } path_r$. By cases on the final rule used. For REL-ENV, the lemma holds because the only direct relations in \mathcal{E} are between variables. All of the remaining rules REL-TRANS, EQ-REL,

HB-SUB-RIGHT, and HB-SUB-LEFT hold by induction. We spell out the argument only for HB-SUB-LEFT: by the inductive hypothesis, the paths $path_c$, $path_p$, and $path$ are each equatable with a variable x_c , x_p , and x . Furthermore, $\mathcal{E} \vdash x_c \text{ subOf } x_p$ and $\mathcal{E} \vdash x_p \text{ hb } x$. This allows us to conclude that $\mathcal{E} \vdash x_c \text{ hb } x$.

Note that most rules (REL-WILDCARD, RBY-WBY, etc) do not apply to this lemma because they only concern one of the *wcrel* relations `permitsWr`, `permitsRd`, `ensuresImm`, or `eq`.

Lemma 5 (Equatable Relations).

Given \mathcal{E} , $path_l$, $path_r$, x_l , x_r , and rel such that:

1. $\vdash \mathcal{E}$
2. $\mathcal{E} \vdash path_l \text{ rel } path_r$.
3. $\mathcal{E} \vdash path_l \text{ eq } x_l$
4. $\mathcal{E} \vdash path_r \text{ eq } x_r$
5. rel is not a transitive relation `trel`

Then there exists a proof that $\mathcal{E} \text{ eq } \vdash x_l \text{ rel } x_r$, where $\mathcal{E} \text{ eq}$ is equal to \mathcal{E} without any `eq` relations.

Proof. By induction on the proof tree that $\mathcal{E} \vdash path_l \text{ rel } path_r$. Proceed by cases on the final rule:

1. REL-ENV We distinguish two cases:
 - If $rel = \text{eq}$, then $path_l$ has the form $x_o.f$ and $path_r = x_r$. By the Uniqueness Lemma, $x_l = x_r$. Rule EQ-SELF assures us that $x_r \text{ eq } x_r$ in any environment, including $\mathcal{E} \text{ eq}$.
 - Otherwise, because $\vdash \mathcal{E}$, the only direct relations in \mathcal{E} are between variables. Therefore, $path_l = x_l$ and $path_r = x_r$, and the proof is trivial.
2. REL-TRANS Does not apply because rel is not a transitive relation.
3. REL-WILDCARD ($wcrel = rel$, $path_2 = path_r$) We distinguish two cases.
 - $rel = \text{eq}$: By the Uniqueness lemma, $x_l = x_r$, and therefore the lemma holds.
 - *Otherwise*: Because $path_l = path_1.f$ is equatable with x_l , the type of $path_l$ must include a type argument equating the ghost field `f`. Therefore, in order to derive a type that uses a different relation, rule T-SUB must have been used. The application of rule T-SUB therefore contains a proof that $path_l \text{ rel } path_r$. By induction the lemma holds.
4. EQ-SELF In this case, $path_l = path_r$. By the Uniqueness lemma, $x_l = x_r$ and therefore the lemma holds.
5. EQ-SYMMETRIC By induction.
6. EQ-EXTEND $path_1$ and $path_2$ must be equatable with variables x_1 and x_2 (otherwise $path_l$ and $path_r$ could not be equated with x_l and x_r). By induction, $x_1 \text{ eq } x_2$ is provable in an environment without `eq` relations, and therefore $x_1 = x_2$. Therefore, $path_l$ and $path_r$ themselves are both equatable with the same value (the field `f` of x_1), and so the lemma holds.
7. EQ-REL By induction and the Uniqueness lemma.

8. HB-SUB-LEFT : By the Variable Relations Only lemma, $\exists x_p. \mathcal{E} \vdash \text{path}_p \text{eq} x_p$. By induction and the Uniqueness Lemma, $\mathcal{E}_{\text{eq}} \vdash x_l \text{ subOf } x_p$ and $\mathcal{E}_{\text{eq}} \vdash x_p \text{ hb } x_r$. These clauses can be used to reapply HB-SUB-LEFT to x_l, x_r in the environment \mathcal{E}_{eq} and so the lemma holds.
9. HB-SUB-RIGHT : Analogous to HB-SUB-LEFT.
10. RBY-WBY : Analogous to HB-SUB-LEFT.
11. RBY-IMMUTABLE : Analogous to HB-SUB-LEFT.
12. WBY-INLINE : Analogous to HB-SUB-LEFT.
13. RBY-SUB : Analogous to HB-SUB-LEFT.
14. WBY-INTERVAL : Analogous to HB-SUB-LEFT.
15. IMM-INTERVAL : Analogous to HB-SUB-LEFT.
16. WBY-LOCK : Analogous to HB-SUB-LEFT.
17. IMM-FINAL : Trivial.

Lemma 6 (Substitution).

Given environments \mathcal{E} and \mathcal{E}' , a statement list lstmts , and variable lists $\mathbf{x}s_d$ and $\mathbf{x}s_v$, such that

- $\mathcal{E} \vdash \text{lstmts} \xrightarrow{\text{fold}} \mathcal{E}_N$
- $[\mathbf{x}s_d \rightarrow \mathbf{x}s_v] \mathcal{E}$ is a sub-environment of \mathcal{E}'
- $(\mathbf{x}s_d \cup \mathbf{x}s_v) \# BV(\text{lstmts})$

then

- $\mathcal{E}' \vdash [\mathbf{x}s_d \rightarrow \mathbf{x}s_v] \text{lstmts} \xrightarrow{\text{fold}} \mathcal{E}'_N$

Proof. The original proof that $\mathcal{E} \vdash \text{lstmts} \xrightarrow{\text{fold}} \mathcal{E}_N$ can be converted into the new proof by replacing any reference to a variable in $\mathbf{x}s_d$ with the corresponding variable from $\mathbf{x}s_v$. All of these judgements will still be valid due to the monotonic nature of our type rules (as expressed in the Monotonic lemma).

5 Guard Lemmas

We begin with the guard lemmas. These lemmas are the heart of our safety proof. They assert that, in a well-formed machine state, a guard only permits writes to one interval at a time. Put another way, any interval which is permitted to write is ordered with respect to all other intervals that are permitted access.

Lemma 7 (Variable Guard (Writes)). Given a well-formed machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$, a guard x_g , and two intervals x, x' where

- \mathcal{E}_{eq} is \mathcal{E} without any eq relations.
- $(x_g : \text{Interval}) \in \mathcal{E}_{\text{eq}} \vee (x_g : \text{Lock}) \in \mathcal{E}_{\text{eq}}$
- $\mathcal{E}_{\text{eq}} \vdash x_g \text{ permitsWr } x$
- $\mathcal{E}_{\text{eq}} \vdash x_g \text{ permitsWr } x'$

then x and x' are ordered in \mathcal{E}_{eq} .

Proof. By induction on the proof trees that both accesses are permitted. We proceed by cases on the rule used to authorize the write. Because only variables are involved and the relation is not transitive, the Equatable Relations Lemma tells us that both proof trees can be derived in an environment without `eq` relations, therefore we assume that the EQ-REL rule was not used.

- REL-ENV In a well-formed environment, `permitsWr` relations are never added directly to the environment, so this rule cannot have been used.
- REL-TRANS `permitsWr` is not a transitive relation, so this rule cannot have been used.
- REL-WILDCARD This rule cannot have been used because it relates a path $path_1.f$, not a single variable x_g .
- EQ-REL Because \mathcal{E}_{eq} contains no `eq` relations, this is only possible if $path'_1 = x_g$ and $path'_2 = x$, in which case the lemma holds by induction.
- WBY-INLINE By the Variable Relations Only lemma, there must exist a variable x_p where $\mathcal{E} \vdash x \text{ inlineSubOf } x_p$. By induction, x_p and x' are ordered. By definition, therefore, x and x_p are ordered.
- WBY-INTERVAL We know that $x_g = x$. We now consider the rules that could have authorized x' to write:
 - REL-ENV, REL-TRANS, REL-WILDCARD Cannot have been used, as before.
 - EQ-REL By induction, as before.
 - WBY-INLINE Holds by induction as before.
 - WBY-INTERVAL $x = x'$ and so they are ordered by definition.
 - WBY-LOCK Cannot occur because x is an interval and not a lock.
- WBY-LOCK We now consider the rules that could have authorized x' to write. We know that x_g is a lock.
 - REL-ENV, REL-TRANS, REL-WILDCARD Cannot have been used, as before.
 - EQ-REL By induction, as before.
 - WBY-INLINE Holds by induction as before.
 - WBY-INTERVAL Cannot occur because x_g is a lock.
 - WBY-LOCK x and x' both hold the lock x_g are therefore ordered by definition.

Lemma 8 (Variable Guard (Reads)). *Given a well-formed machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$, a guard x_g , and two intervals x, x' where*

- \mathcal{E}_{eq} is \mathcal{E} without any `eq` relations.
- $(x_g : \text{Interval}) \in \mathcal{E}_{eq} \vee (x_g : \text{Lock}) \in \mathcal{E}_{eq}$
- $\mathcal{E}_{eq} \vdash x_g \text{ permitsWr } x$
- $\mathcal{E}_{eq} \vdash x_g \text{ permitsRd } x'$

then x is ordered with respect to x' in \mathcal{E}_{eq} .

Proof. Analogous to the Variable Guard (Writes) lemma. Most cases are precisely the same. We show only those that differ:

- WB_Y-INTERVAL We know that $x_g = x$. We now consider the rules that could have authorized x' to read:
 - REL-ENV, REL-TRANS, REL-WILDCARD Cannot have been used, as before.
 - EQ-REL Impossible because \mathcal{E}_{eq} contains no **eq** relations.
 - RB_Y-WB_Y By the Variable Guard (Writes) lemma, x and x' are ordered.
 - RB_Y-IMMUTABLE The only way to prove that $\mathcal{E}_{\text{eq}} \vdash x x'$ is by rule IMM-INTERVAL. This rule requires that $\mathcal{E}_{\text{eq}} \vdash x \text{hb} x'$, and thus the accesses are ordered.
 - RB_Y-SUB By induction we know that x is ordered with respect to the parent interval x_p . By definition it is ordered with respect to x' .
- WB_Y-LOCK We now consider the rules that could have authorized x' to read. We know that x_g is a lock.
 - REL-ENV, REL-TRANS, REL-WILDCARD Cannot have been used, as before.
 - EQ-REL Impossible because \mathcal{E}_{eq} contains no **eq** relations.
 - RB_Y-WB_Y By the Variable Guard (Writes) lemma, x and x' are ordered.
 - RB_Y-IMMUTABLE Cannot occur because x_g is a lock and no rule will prove that a lock is immutable.
 - RB_Y-SUB By induction we know that x is ordered with respect to the parent interval x_p . By definition it is ordered with respect to x' .

Lemma 9 (Path Guard). *Given a well-formed machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$, a guard $path_g$, and two intervals x, x' where*

- $\mathcal{E} \vdash path_g \text{permitsWr } x$
- $\mathcal{E} \vdash path_g \text{permitsRd } x'$

then x and x' are ordered in \mathcal{E} .

Proof. We consider two cases:

1. $\exists x_g. \mathcal{E} \vdash path_g \text{eq } x_g$: By the Equatable Relations Lemma, and because $(\mathcal{E}; \mathcal{O}; \mathcal{A})$ is well-typed, we know that $\mathcal{E}_{\text{eq}} \vdash x_g \text{permitsWr } x$ and that $\mathcal{E}_{\text{eq}} \vdash x_g \text{permitsRd } x'$. By the Variable Guard (Reads) Lemma, x and x' are ordered with respect to \mathcal{E}_{eq} . As $\mathcal{E}_{\text{eq}} \subseteq \mathcal{E}$, they must also be ordered in \mathcal{E} .
2. No equatable variable x_g exists. We proceed by induction on the proof three that $\mathcal{E} \vdash path_g \text{permitsWr } x$. As all relations in \mathcal{E} are between local variables, this proof tree must end with an application of REL-WILDCARD, where $w\text{crel}$ is **permitsWr**, $path_g = path_1.f$, and $path_2 = x$. The typing of $path_1$ could be achieved through two rules:
 - T-FIELD : The Path Type lemma states that ty is unique and invariant. Therefore, $path_g$ can only permit writes from x or its inline subintervals (via WB_Y-INLINE). We will now examine each way that x' could have been permitted to read and show that, in each case, x' must be ordered with respect to x :

- RBY-WBY As $path_g$ only permits writes from x or its inline subintervals, x' must either equal x or be an inline subinterval of x . In either case, the two intervals are ordered by definition.
- RBY-IMMUTABLE Could not have been used as there is no way to prove `ensuresImm` unless one can prove `hb`, which requires a variable guard per the Variable Relations Only lemma.
- RBY-SUB By induction and by definition of ordering.
- T-SUB : The proof tree for T-SUB contains a proof that $\mathcal{E} \vdash path_1.fpermitsWrx$. The lemma is therefore true by induction.

6 Preservation Lemma

We now present a preservation lemma. The lemma states that a well-formed machine remains well-formed as it progresses. It is mostly *de rigueur*. Note that we do not prove a progress lemma: there are a number of ways that the machine can get stuck — such as null pointer errors, assertion failures, or deadlocks — which our type system does not attempt to prevent.

Lemma 10 (Preservation).

Given two machine states $(\mathcal{E}; \mathcal{O}; \mathcal{A})$ and $(\mathcal{E}'; \mathcal{O}'; \mathcal{A}')$ such that

- $\vdash (\mathcal{E}; \mathcal{O}; \mathcal{A})$
- $(\mathcal{E}; \mathcal{O}; \mathcal{A}) \rightarrow (\mathcal{E}'; \mathcal{O}'; \mathcal{A}')$

then $\vdash (\mathcal{E}'; \mathcal{O}'; \mathcal{A}')$.

Proof. We proceed by cases on the step which was taken.

- STEP-START Adding a start point to \mathcal{O} can only invalidate its well-formedness if there are intervals which *happen before* x but which have not yet ended. This cannot be case because x was judged to be ready.
- STEP-MID Adding a mid point to \mathcal{O} can invalidate its well-formedness if (a) the interval has not yet started or (b) the interval has inline subintervals that not yet ended. The preconditions of STEP-MID exclude both possibilities.
- STEP-END Adding an end point to \mathcal{O} can invalidate its well-formedness if (a) the mid point of the interval has not yet occurred or (b) there are children which have not yet ended. The preconditions of STEP-END exclude both possibilities.
- STEP-INLINE We proceed by cases on the EXEC rule which was used. We also note that because $\vdash (\mathcal{E}; \mathcal{O}; \mathcal{A})$, there exists a proof tree that $stable(\mathcal{E}, x_l) \vdash lstmts \xrightarrow{fold} \mathcal{E}_N$, where x_l is the label of the first statement $lstmt = x_l : \dots$ in $lstmts$. Let $lstmts_r$ be the remaining statements.
 - EXEC-STORE : We know that
 - * $lstmt = x_l : (x_o.f = x_v)$
 - * $\mathcal{E}' = (\mathcal{E} \setminus x_o.f) + (x_o.f \text{ eq } x_v)$

The only change from the original machine state is that \mathcal{E}' equates $\mathbf{x}_o.f$ is equated with a new value. Because the new value has the right type, we know that \mathcal{E}' is still well-formed.

$\mathcal{E}' \vdash \mathcal{O}$ still holds because \mathcal{E}' did not add any **hb** relations.

$\mathcal{E}' \vdash \mathcal{A}$ still holds because the new environment \mathcal{E}' differs from \mathcal{E} only by the tuple equating $\mathbf{x}_o.f$. This tuple cannot have been used in the original type check however because it would have been filtered out by the function *stable()*.

- EXEC-LOAD EXEC-LOAD does not affect the machine state except to substitute the value \mathbf{x}_v that was loaded for the variable \mathbf{x}_d (the result of the load) in the remaining statements. We show that these remaining statements will still type check by examining the type check rule that was used to type the load:

* STMT-LOAD : Because the initial state was well-formed, we know that:

$$\begin{aligned} & \cdot \mathcal{E} + (\mathbf{x}_d : \mathbf{ty}_f) \vdash \mathit{lstmts}_r \xrightarrow{\mathit{fold}} \mathcal{E}_N \\ & \cdot \mathcal{E} \vdash \mathbf{x}_v : \mathbf{ty}_f \\ & \cdot \mathbf{x}_d \notin BV(\mathcal{E}) \end{aligned}$$

Therefore by the Substitution lemma, the type check for lstmts_r can be converted to a type check for $[\mathbf{x}_d \rightarrow \mathbf{x}_v]\mathit{lstmts}_r$.

* STMT-LOAD-IMMUTABLE : Because the initial state was well-formed, we know that:

$$\begin{aligned} & \cdot \mathcal{E} + (\mathbf{x}_d : \mathbf{ty}_f) + (\mathbf{x}_o.f \mathbf{eq} \mathbf{x}_d) \vdash \mathit{lstmts}_r \xrightarrow{\mathit{fold}} \mathcal{E}_N \\ & \cdot \mathcal{E} \vdash \mathbf{x}_v : \mathbf{ty}_f \\ & \cdot (\mathbf{x}_v \mathbf{eq} \mathbf{ty}_f) \in \mathcal{E} \\ & \cdot \mathbf{x}_d \notin BV(\mathcal{E}) \end{aligned}$$

Therefore by the Substitution lemma, the type check for lstmts_r can be converted to a type check for $[\mathbf{x}_d \rightarrow \mathbf{x}_v]\mathit{lstmts}_r$.

- EXEC-NEW EXEC-NEW affects the new machine state in three ways:
 1. It adds a new object to the environment along with values for its fields and relations among its members. These new relations can be shown to be well-formed by inspection of the *initial()* function. We know that $\mathcal{E}' \vdash \mathcal{O}$ because of the conditions enforced by the *initial()* function: in particular, it does not add **hb** relations to any interval that could already have started.
 2. It replaces the variable \mathbf{x}_d with the newly created variable \mathbf{x}_n in all subsequent statements. This is safe for the same reasons it is safe in a load: the only fact known about \mathbf{x}_d during the original type check was its type, and \mathbf{x}_n has the same type (along with other additional relations).
 3. It creates asynchronous intervals for every interval member. These new additions to \mathcal{A} are known to type check because the bodies of each interval member were type-checked individually in the context of their class definition. The environment in the class (after substitution for **this**) is a subset of the tuples created by *initial()*. Therefore, by the Substitution lemma, the new interval bodies will type check.

- EXEC-METHOD

We now show that the new list of statements after inlining the method body will type check. We proceed in two steps. We first show that the imported statements $lstmts_m$ are typable in their new context. Because the method was type-checked individually, we know that it is checkable in a context containing

1. the class relations ($\mathbf{this} : c$, etc;
2. the method relations (definitions for `method` and the parameters, $reqs_m$); and,
3. relations pertaining to the labels in $lstmts_m$.

All of these relations (modulo substitution) must exist in \mathcal{E} :

- * The class relations must exist because the receiver is not null and the state is well-formed.
- * The method relations must exist because the type check passed.
- * The relations for the labels in $lstmts_m$ are created by the EXEC-METHOD rule itself.

Therefore, by the Substitution lemma, the statements are still typable.

Finally, we must show that the statements $lstmts_r$ which follow the method call will still be checkable now that the method body has been inlined. This follows because checking the method body can only grow the environment that is used to check $lstmts_r$, and we know that $lstmts_r$ were checkable in the original environment \mathcal{E} .

- EXEC-ASSERT

The type check for the subsequent statements may require that the assert condition holds. Because the EXEC-ASSERT rule fired, we know that the assertion condition holds in \mathcal{E} . Therefore, the remaining statements will still type-check after the assertion is removed.

7 Data Race Theorem

Theorem 1 (Data Race Freedom).

For any execution beginning from a well-formed initial machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$, every write is ordered with respect to all other accesses.

Proof. Because the initial state is well-formed, we know that all subsequent states are well-formed by the Preservation lemma. Because each state is well-formed, every field has a unique, well-defined guard path, and all load and stores must be judged to be permitted by that path. By the Path Guard lemma, the intervals containing those loads and stores must be ordered.