Online Appendix to: Types for Atomicity: Static Checking and Inference for Java

CORMAC FLANAGAN
University of California at Santa Cruz
STEPHEN N. FREUND and MARINA LIFSHIN
Williams College
and
SHAZ QADEER
Microsoft Research

We present in this appendix the formal development for AtomicJava and our type inference algorithm. We begin with the semantics of AtomicJava in Appendix A and present additional details about the type system in Appendix B.

Appendix C shows the key property of our type system: atomic blocks in well-typed AtomicJava programs are serializable, based on Lipton's theory of reduction [Lipton 1975].

The remaining appendices show auxiliary properties that are needed in the key serializability proof. Appendix D shows that evaluation preserves typing, and Appendix E describes when a thread has exclusive access to a field. Finally, Appendix F shows that the type inference algorithm is sound with respect to the type rules.

For reference, we include indexes of the symbols and judgments used throughout this Appendix in Tables III and IV.

A. FORMAL SEMANTICS

We specify the operational semantics of AtomicJava using the abstract machine in Figure 16. The machine evaluates a program by stepping through a sequence of states. A state consists of two components: an object store and a sequence of expressions, each of which is a thread. New threads are added to the end of the sequence. We use $T.T'$ to denote the concatenation of two sequences. Each thread is given a unique thread identifier, or *thread id*, which is the thread's position in the sequence.

To express intermediate runtime states, we extend the AtomicJava syntax in the following ways. (These new constructs should not appear in source programs.)

Table III. Symbols

Symbol	Page	Meaning
$\{\!\mid db \mid\!\}^o_c$	49	object record
$[\![d]\!]$	28	meaning function mapping closed atomicity expressions to atomicities
$\alpha(E, e)$	60	atomicity of an expression e in environment E.
Π	49	program state
ρ	49	object address
σ	49	object store
θ	13	substitution
A	28	atomicity assignment
a	10	atomicity (either basic or conditional)
b	10	basic atomicity
$B(t)$	15	atomicity of a read/write to field of type t
C	27	atomicity constraint $d \sqsubseteq s$
c	7	class type
ci	49	class instantiation
cn	7	class name
db	49	object field value map
d	27	syntactic atomicity expression
$\mathcal{E}$	49	evaluation context
e	7	expression
E	12	typing environment
fd	7	field name
g	7	field guard
$IS(l, a)$	56	atomicity of `in-sync` l e, where e has atomicity a
$lift(P, E, d)$	27	atomicity expression to lift meaning of d to be valid in environment E
l	7	lock expression
ls	57	lock set
md	7	method name
n	7	number
o	49	object lock state
P	7	program
$R(a)$	18	atomicity of `assert-atomic` e, where e has atomicity a
$\mathcal{R}(d)$	27	atomicity expression for `assert-atomic` e, where d is the atomicity expression for e
$\mathcal{S}(l, d)$	27	atomicity expression for `synchronized` l e, where d is the atomicity expression for e
$S(l, a)$	17	atomicity of `synchronized` l e, where e has atomicity a
s	26	open atomicity (ie, atomicity a or atomicity variable α)
T	49	thread sequence
t	7	type
v	7	value
x, y	7	variable
$Y(\mathcal{E}, a)$	60	atomicity a, simplified with respect to the locks held in context $\mathcal{E}$

—Values now include the special value **wrong**, which indicates that a thread has gone wrong by dereferencing `null`. (This construct will occur only at the top-level in a thread.)

—Values also include *addresses*, ranged over by the meta-variable ρ.

—Expressions now include the `in-sync` and `in-atomic` constructs. The construct `in-sync` ρ e indicates that e is being evaluated while holding the lock

Table IV. Logical Relations and Judgments

Form	Page	Meaning
$a_1 \sqsubseteq a_2$	10	ordering relation for atomicities
$a_1 \equiv a_2$	10	semantic equivalence for atomicities
$P; E \vdash e : t \cdot a$	14	expression e has type t and atomicity a in environment E
$P \vdash E$	16	environment E is well-formed
$P; E \vdash t$	16	type t is well-formed
$P; E \vdash a$	16	atomicity a is well-formed
$P; E \vdash_{\text{lock}} l$	16	expression l is a well-formed lock expression
$P; E \vdash a \uparrow a'$	16	atomicity a' is the well-formed smallest atomicity greater than or equal to a
$P; E \vdash \mathit{field}$	16	*field* is well-formed
$P; E \vdash \mathit{meth}$	16	*meth* is well-formed
$P; E \vdash \mathit{defn}$	16	*defn* is well-formed
$P \vdash \text{wf}$	16	P is well-formed
$A \models C$	28	constraint C is satisfied by assignment A
$P; E \vdash e : t \cdot d \cdot \bar{C}$	29	expression e has type t and atomicity expression d in environment E under any assignment satisfying $\bar{C}$
$P \vdash E \cdot \bar{C}$	31	environment E is well-formed under any assignment satisfying $\bar{C}$
$P; E \vdash t \cdot \bar{C}$	31	type t is well-formed under any assignment satisfying $\bar{C}$
$P; E \vdash s \cdot \bar{C}$	31	open atomicity s is well-formed under any assignment satisfying $\bar{C}$
$P; E \vdash_{\text{lock}} l \cdot \bar{C}$	31	expression l is a well-formed lock expression under any assignment satisfying $\bar{C}$
$P; E \vdash \mathit{field} \cdot \bar{C}$	31	*field* is well-formed under any assignment satisfying $\bar{C}$
$P; E \vdash \mathit{meth} \cdot \bar{C}$	31	*meth* is well-formed under any assignment satisfying $\bar{C}$
$P \vdash \mathit{defn} \cdot \bar{C}$	31	*defn* is well-formed under any assignment satisfying $\bar{C}$
$P \vdash \bar{C}$	31	P is well-formed under any assignment satisfying $\bar{C}$
$A \rightarrow^{\bar{C}} A'$	33	one step in the iterative constraint solver for $\bar{C}$
$A \rightarrow^{\bar{C}} \text{Error}$	33	constraints $\bar{C}$ are not satisfiable by A or any larger assignment
$a \rightarrow^{h}_{n} a'$	36	atomicity a can be simplified to a' if the locks in h are held and the locks in n are not held
$P \vdash \Pi \rightarrow_i \Pi'$	49	semantics for thread i
$P \vdash \Pi \rightarrow \Pi'$	49	standard program semantics
$P \vdash \Pi \mapsto \Pi'$	49	serialized program semantics
$P; E; \rho \vdash \mathit{obj}$	55	object obj is well-formed
$P; E \vdash \sigma$	55	store σ is well-formed
$P \vdash \Pi$	55	state Π is well-formed
$\mathit{ls} \vdash_{\text{cs}} e$	57	lock set ls contains all locks held in e
$\vdash_{\text{cs}} \Pi$	57	no two threads in state Π hold the same lock

ρ. The construct `in-atomic` e indicates that e is being evaluated and that its atomicity should be at most `atomic`.

—The grammar now includes *class instantiation*s, which are simply instantiations of a parameterized class.

Objects are kept in an object store σ that maps addresses to objects. An object $\{\!|\, db \,|\!\}^{o}_{c}$ has three components: the class c of the object, a field map db mapping field names to values, and a lock state o. A field map db is typically written as a list $fd_1 = v_1, \ldots, fd_n = v_n$, where fd_i maps to the value v_i. The lock state o is

State space

$$
\begin{array}{rcl}
\Pi & \in & \mathit{State} = \mathit{Store} \times \mathit{ThreadSeq} \\
\sigma & \in & \mathit{Store} = \rho \rightharpoonup d \\
T & \in & \mathit{ThreadSeq} = e^* \\
\mathit{obj} & ::= & \{\!|\, \mathit{db} \,|\!\}^o_c \\
\mathit{db} & ::= & \mathit{fd}_1 = v_1, \cdots, \mathit{fd}_n = v_n \\
o & ::= & \{\bot, 1, 2, \ldots\}
\end{array}
$$

Grammar extensions

$$
\begin{array}{rcl}
v & ::= & \ldots \mid \rho \mid \textbf{wrong} \\
e & ::= & \ldots \mid \texttt{in-sync}\ \rho\ e \mid \texttt{in-atomic}\ e \\
 & & \\
ci & ::= & \texttt{class}\ c\ \mathit{body} \qquad\qquad\qquad \text{(class instantiation)}
\end{array}
$$

Evaluation contexts

$$
\begin{array}{rcl}
\mathcal{E} & ::= & [\,] \mid \texttt{new}_y\ c\texttt{(}v^*, \mathcal{E}, e^*\texttt{)} \mid \mathcal{E}.\mathit{fd} \mid \mathcal{E}.\mathit{fd}\ \texttt{=}\ e \mid \rho.\mathit{fd}\ \texttt{=}\ \mathcal{E} \\
 & \mid & \mathcal{E}.\mathit{md}\texttt{(}e^*\texttt{)} \mid \rho.\mathit{md}\texttt{(}v^*, \mathcal{E}, e^*\texttt{)} \mid \texttt{let}\ x\ \texttt{=}\ \mathcal{E}\ \texttt{in}\ e \\
 & \mid & \texttt{if}\ \mathcal{E}\ e\ e \mid \mathcal{E}\texttt{.fork} \\
 & \mid & \texttt{sync}\ \mathcal{E}\ e \mid \texttt{in-sync}\ \rho\ \mathcal{E} \mid \texttt{in-atomic}\ \mathcal{E}
\end{array}
$$

Transition rules

$$
\begin{array}{lll}
(\textit{standard semantics}) & P \vdash \langle \sigma, T \rangle \ \rightarrow\ \langle \sigma', T' \rangle & \text{if } P \vdash \langle \sigma, T \rangle \rightarrow_i \langle \sigma', T' \rangle \\
 & & \\
(\textit{serial semantics}) & P \vdash \langle \sigma, T \rangle \ \mapsto\ \langle \sigma', T' \rangle & \text{if } P \vdash \langle \sigma, T \rangle \rightarrow_i \langle \sigma', T' \rangle \\
 & & \text{and } \neg\exists j.\ (i \neq j) \wedge (T_j \equiv \mathcal{E}[\texttt{in-atomic}\ e])
\end{array}
$$

Fig. 16. The semantics for AtomicJava.

either in the unlocked or locked, which are denoted by $\bot$ and the thread id of the thread holding the lock, respectively.

We use $\sigma[\rho \mapsto d]$ to denote the store that agrees with σ at all addresses except ρ, which is mapped to d. The store $\sigma[\rho.\mathit{fd} \mapsto v]$ denotes the store that agrees with σ at all addresses except ρ, which is mapped to the record $\sigma(\rho)$ updated so that field *fd* contains value v.

Program evaluation begins in a state with an empty store $\emptyset$ and a single thread. Evaluation proceeds according to the *standard* transition relation $\rightarrow$, which arbitrarily interleaves steps of the various threads. This relation leverages the transition relation $\rightarrow_i$ for individual thread steps from Figure 17. The rules in that figure define the semantics of thread i in terms of evaluation contexts. An evaluation context $\mathcal{E}$, as shown in Figure 16, is an expression containing a hole [ ] in place of the next subexpression to be evaluated. Figure 17 contains a rule for each possible expression appearing in the hole of the evaluation context for thread i. A program terminates when all threads have been reduced to values.

We also define a second *serial* transition relation $\mapsto$, in which atomic blocks are executed without interleaved steps from other threads. We show below that these two semantics are equivalent for well-typed programs.

[RED NEW]
$$P \vdash \langle \sigma, T.\mathcal{E}[\mathtt{new}_y\ c(v_{1..n})].T' \rangle \quad \rightarrow_i \quad \langle \sigma[\rho \mapsto \{\!|\, fd_i = v_i\ ^{i \in 1..n}\, |\!\}^{\perp}_c], T.\mathcal{E}[\rho].T' \rangle$$
$$\text{if } \rho \notin dom(\sigma) \text{ and } P \vdash_{\text{inst}} \mathtt{class}\ c\ \{\ t_i\ fd_i\ g_i\ ^{i \in 1..n}\ \ldots\ \}$$

[RED READ]
$$P \vdash \langle \sigma, T.\mathcal{E}[\rho.fd].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[v].T' \rangle \quad \text{if } \sigma(\rho) = \{\!|\, \ldots, fd = v, \ldots |\!\}^o_c$$

[RED WRITE]
$$P \vdash \langle \sigma, T.\mathcal{E}[\rho.fd\ \mathtt{=}\ v].T' \rangle \quad \rightarrow_i \quad \langle \sigma[\rho.fd \mapsto v], T.\mathcal{E}[v].T' \rangle$$
$$\text{if } \sigma(\rho) = \{\!|\, \ldots, fd = v', \ldots |\!\}^o_c$$

[RED INVOKE]
$$P \vdash \langle \sigma, T.\mathcal{E}[\rho.md\langle l_{1..k}\rangle\mathtt{(}v_{1..r}\mathtt{)}].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[\theta(e)].T' \rangle$$
$$\text{if } \sigma(\rho) = \{\!|\, db\, |\!\}^o_c$$
$$\text{and } P \vdash_{\text{inst}} \mathtt{class}\ c\ \{\ \ldots\ a\ t\ md\langle \mathtt{ghost}\ y_{1..k}\rangle\mathtt{(}t_i\ z_i\ ^{i \in 1..r}\mathtt{)}\ e\ \ldots\ \}$$
$$\text{and } \theta = [\mathtt{this} := \rho, y_i := l_i\ ^{i \in 1..k}, z_i := v_i\ ^{i \in 1..r}]$$

[RED LET]
$$P \vdash \langle \sigma, T.\mathcal{E}[\mathtt{let}\ x\ \mathtt{=}\ v\ \mathtt{in}\ e].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[e[x := v]].T' \rangle$$

[RED IF-TRUE]
$$P \vdash \langle \sigma, T.\mathcal{E}[\mathtt{if}\ v\ e_2\ e_3].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[e_2].T' \rangle \quad \text{if } v \neq 0$$

[RED IF-FALSE]
$$P \vdash \langle \sigma, T.\mathcal{E}[\mathtt{if}\ 0\ e_2\ e_3].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[e_3].T' \rangle$$

[RED WHILE]
$$P \vdash \langle \sigma, T.\mathcal{E}[\mathtt{while}\ e_1\ e_2].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[\mathtt{if}\ e_1\ (e_2;\ \mathtt{while}\ e_1\ e_2)\ 0].T' \rangle$$

[RED FORK]
$$P \vdash \langle \sigma, T.\mathcal{E}[\rho.\mathtt{fork}].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[0].T'.e \rangle$$
$$\text{where } e = (\mathtt{let}\ x\ \mathtt{=}\ \mathtt{new}\ \mathtt{Object()}\ \mathtt{in}\ \mathtt{sync}\ x\ (\rho.\mathtt{run}\langle x\rangle\mathtt{()}))$$

[RED SYNC]
$$P \vdash \langle \sigma[\rho \mapsto \{\!|\, db\, |\!\}^{\perp}_c], T.\mathcal{E}[\mathtt{sync}\ \rho\ e].T' \rangle \quad \rightarrow_i \quad \langle \sigma[\rho \mapsto \{\!|\, db\, |\!\}^{i}_c], T.\mathcal{E}[\mathtt{in\text{-}sync}\ \rho\ e].T' \rangle$$

[RED RE-SYNC]
$$P \vdash \langle \sigma[\rho \mapsto \{\!|\, db\, |\!\}^{i}_c], T.\mathcal{E}[\mathtt{sync}\ \rho\ e].T' \rangle \quad \rightarrow_i \quad \langle \sigma[\rho \mapsto \{\!|\, db\, |\!\}^{i}_c], T.\mathcal{E}[e].T' \rangle$$

[RED IN-SYNC]
$$P \vdash \langle \sigma[\rho \mapsto \{\!|\, db\, |\!\}^{i}_c], T.\mathcal{E}[\mathtt{in\text{-}sync}\ \rho\ v].T' \rangle \quad \rightarrow_i \quad \langle \sigma[\rho \mapsto \{\!|\, db\, |\!\}^{\perp}_c], T.\mathcal{E}[v].T' \rangle$$

[RED ATOMIC]
$$P \vdash \langle \sigma, T.\mathcal{E}[\mathtt{assert\text{-}atomic}\ e].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[\mathtt{in\text{-}atomic}\ e].T' \rangle$$

[RED IN ATOMIC]
$$P \vdash \langle \sigma, T.\mathcal{E}[\mathtt{in\text{-}atomic}\ v].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathcal{E}[v].T' \rangle$$

[RED WRONG]
$$P \vdash \langle \sigma, T.\mathcal{E}[e].T' \rangle \quad \rightarrow_i \quad \langle \sigma, T.\mathbf{wrong}.T' \rangle$$
$$\text{if } e \in \{\mathtt{null}.fd,\ \mathtt{null}.fd\ \mathtt{=}\ v,\ \mathtt{null}.md\mathtt{(}v^*\mathtt{)}, \mathtt{null.fork},$$
$$\mathtt{sync}\ \mathtt{null}\ e',\ \mathtt{in\text{-}sync}\ \mathtt{null}\ v\}$$

Fig. 17. Transition rules for AtomicJava (where $i = |T| + 1$).

The reduction rules for $\rightarrow_i$ are mostly straightforward. The rule [RED NEW] allocates a new object record and stores the provided initial values into the object's fields. This rule refers to the auxiliary judgment $P \vdash_{\text{inst}} ci$, which indicates that ci is a valid instantiation of a class definition:

$$\text{[CLASS INST]}\quad \frac{\texttt{class}\ cn\langle\texttt{ghost}\ x_i{}^{i\in 1..n}\rangle\ body \in P}{P \vdash_{\text{inst}} \texttt{class}\ cn\langle l_{1..n}\rangle\ body[x_i := l_i{}^{i\in 1..n}]}$$

The rule [RED SYNC] reduces the expression `sync` ρ `in` e to `in-sync` ρ e by acquiring the lock of ρ. After e evaluates to some value v, the rule [RED IN-SYNC] releases the lock of ρ and returns the value v. The rule [RED RE-SYNC] permits re-entrant locks. In essence, if a lock is already held by the current thread, attempting to reacquire it is a "no-op."

The rule [RED FORK] for ρ`.fork` creates a new thread to evaluate ρ`.run()`, and returns 0 as the (dummy) result of the fork expression. This rule allocates a fresh object x for use as the thread-local lock for the new thread. The lock for x is acquired before invoking the `run` method. We assume that every program contains an empty class declaration for `Object`.

The rule [RED ATOMIC] reduces the expression `assert-atomic` e to `in-atomic` e to mark that e is being evaluated, at which stage the serialized semantics avoids scheduling concurrent threads. Once e reduces to some value v, `in-atomic` v reduces to v via the rule [RED IN-ATOMIC].

If a thread dereferences `null`, it goes to the special expression **wrong** via the rule [RED WRONG], but other threads may still continue to execute.

We define race conditions (or conflicting accesses) in terms of the semantics as follows. An expression e *accesses* a field $\rho.fd$ if $e = \mathcal{E}[\rho.fd]$ or $e = \mathcal{E}[\rho.fd = v]$ for some $\mathcal{E}$ and v. A state has *conflicting accesses* on $\rho.fd$ if its thread sequence contains two or more expressions that access $\rho.fd$ and at least one of the accesses is a write access. Also, an expression e is *in a critical section on* ρ if $e = \mathcal{E}[\texttt{in-sync}\ \rho\ e']$ for some evaluation context $\mathcal{E}$ and expression e'.

B. TYPE SYSTEM

B.1 Dependent Types and Evaluation

Lock expressions appearing in dependent types complicate the soundness proof, because they require reasoning about semantic equivalence of lock expressions. For example, consider the following class definitions:

```
class A {
   final B f;
}

class B {
   final C⟨this⟩ g;
}
```

If the address ρ has type A (i.e., $\sigma(\rho) = \{\!|\, \texttt{f} = \rho' \,|\!\}^{o}_{\texttt{A}}$), then the expression $\rho.\texttt{f}.\texttt{g}$ has type $\texttt{C}\langle\rho.\texttt{f}\rangle$. However, $\rho.\texttt{f}.\texttt{g}$ evaluates to $\rho'.\texttt{g}$, which has type $\texttt{C}\langle\rho'\rangle$. Thus, proving type soundness requires proving that $\rho.\texttt{f}$ and ρ' are semantically equivalent with respect to the program's store σ, perhaps via a rule such as the following:

$$\text{[EQUIV EXP]}\quad \frac{P \vdash \langle\sigma, e_i\rangle \rightarrow^* \langle\sigma, v\rangle \qquad \forall i \in 1..2}{P \vdash_\sigma e_1 \leftrightarrow e_2}$$

We could then extend the notion of semantic equivalence $\leftrightarrow$ from expressions to types and atomicities.

These issues regarding dependent types are fairly well understood (e.g., see Cardelli [1988]), but they substantially increase the length and complexity of the type soundness proof. So that we may focus on the novel aspects of our type system, we restrict the type system as follows to avoid this additional complexity in the soundness proof.

RESTRICTION 13. *The only valid lock expressions are ghost variables and values, as explicated by the following restricted version of the rule* [LOCK EXP]*:*

$$\text{[LOCK EXP]}\quad \frac{P; E \vdash v : c \cdot \texttt{const}}{P; E \vdash_{\text{lock}} v}$$

In effect, this restriction ensures that the only expressions appearing inside types are variables (and addresses). Thus, we do not need to consider situations in which evaluation changes an expression's type.

B.2 Left Movers

The `in-sync` $\rho\, e$ construct includes an implicit lock release, which is a left mover. To accommodate this construct in our type system, we introduce an additional basic atomicity `left`. (We could also introduce the symmetric the notion of a right mover to model lock acquires as in Flanagan and Qadeer [2003b], but this is not necessary for our formal development due to the block structure of synchronized statements.) We extend the sequential composition and iterative closure operations to include this new basic atomicity:

b	b^*
const	const
mover	mover
left	left
atomic	cmpd
cmpd	cmpd
error	error

;	const	mover	left	atomic	cmpd	error
const	const	mover	left	atomic	cmpd	error
mover	mover	mover	left	atomic	cmpd	error
left	left	left	left	cmpd	cmpd	error
atomic	atomic	atomic	atomic	cmpd	cmpd	error
cmpd	cmpd	cmpd	cmpd	cmpd	cmpd	error
error	error	error	error	error	error	error

Our partial order on basic atomicities is extended to include `left` as well.

$$\texttt{const} \sqsubset \texttt{mover} \sqsubset \texttt{left} \sqsubset \texttt{atomic} \sqsubset \texttt{cmpd} \sqsubset \texttt{error}$$

$$\boxed{P; E \vdash e : t \cdot a}$$

[EXP ADDR]
$$\frac{P \vdash E \qquad E = E_1, c\ \rho, E_2}{P; E \vdash \rho : c \cdot \texttt{const}}$$

[EXP IN-SYNC]
$$\frac{P; E \vdash_{\text{lock}} \rho \qquad P; E \vdash e : t \cdot a}{P; E \vdash \texttt{in-sync}\ \rho\ e : t \cdot IS(\rho, a)}$$

[EXP IN-ATOMIC]
$$\frac{P; E \vdash e : t \cdot a}{P; E \vdash \texttt{in-atomic}\ e : t \cdot R(a)}$$

[EXP WRONG]
$$\frac{P; E \vdash t}{P; E \vdash \textbf{wrong} : t \cdot \texttt{const}}$$

$$\boxed{P \vdash E}$$

[ENV P]
$$\frac{P; E \vdash c \qquad \rho \notin dom(E)}{P \vdash (E, c\ \rho)}$$

$$\boxed{P; E; \rho \vdash obj}$$

[OBJECT]
$$\frac{P; E \vdash c \qquad P \vdash_{\text{inst}} \texttt{class}\ c\ \texttt{\{}\ t_i\ fd_i\ g_i\ ^{i \in 1..n}\ meth_{1..k}\ \texttt{\}} \qquad P; E \vdash v_i : t_i[\texttt{this} := \rho] \cdot \texttt{const} \quad \forall i \in 1..n}{P; E; \rho \vdash \{\!| fd_i = v_i\ ^{i \in 1..n} |\!\}_c^o}$$

$$\boxed{P; E \vdash \sigma}$$

[STORE]
$$\frac{dom(\sigma) = \{\rho_1, \ldots, \rho_n\} \qquad \sigma(\rho_i) = \{\!| db_i |\!\}_{c_i}^{o_i} \quad \forall i \in 1..n \qquad E = c_1\ \rho_1, \ldots, c_n\ \rho_n \qquad P; E; \rho_i \vdash \sigma(\rho_i) \quad \forall i \in 1..n}{P; E \vdash \sigma}$$

$$\boxed{P \vdash \Pi}$$

[STATE]
$$\frac{P; E \vdash \sigma \qquad P; E \vdash e_i : t_i \cdot a_i \quad \forall i \in 1..|T| \qquad a_i \sqsubseteq \texttt{cmpd} \quad \forall i \in 1..|T|}{P \vdash \langle \sigma, T \rangle}$$

Fig. 18. AtomicJava type rules for runtime terms.

We also define S for `left`:

$$S(l, \texttt{left}) = l\ ?\ \texttt{left:atomic}$$

B.3 Well-Typed Run-Time States

We now extend the AtomicJava type system to run-time states via the rules presented in Figure 18. Environments may now also contain addresses:

$$E ::= \epsilon \mid E, t\ x \mid E, \texttt{ghost}\ x \mid E, c\ \rho$$

The rules [OBJECT] and [STORE] ensure that an object store is well-typed, taking care to properly handle references to self in field types. The rule [STATE] for a program state $\langle \sigma, T \rangle$ ensures that σ is well-typed and that all threads in T are well-typed and have a non-`error` atomicity. We also introduce rule [EXP ADDR] to assign types to addresses appearing in the environment and the rule [EXP WRONG] to assign any type to the value **wrong**.

The most interesting expression type rule is the rule [EXP IN-SYN] for `in-sync` l `e`. This rule uses the following function IS to characterize the atomicity of executing both e and the implicit lock release at the end of the synchronized block. It is defined as follows:

$$\begin{aligned} IS(l, \texttt{const}) &= \texttt{left} \\ IS(l, \texttt{mover}) &= \texttt{left} \\ IS(l, \texttt{left}) &= \texttt{left} \\ IS(l, \texttt{atomic}) &= \texttt{atomic} \end{aligned}$$

$$
\begin{aligned}
IS(l, \texttt{cmpd}) &= \texttt{cmpd}\\
IS(l, \texttt{error}) &= \texttt{error}\\
IS(l, (l\,?\,a_1 : a_2)) &= IS(l, a_1)\\
IS(l, (l'\,?\,a_1 : a_2)) &= l'\,?\,IS(l, a_1) : IS(l, a_2) \ \text{ if } l \neq l'
\end{aligned}
$$

Similarly, the rule [EXP IN-ATOMIC] for an expression `in-atomic` e ensures that expression e has atomicity no greater than `atomic`, using the same technique as the rule [EXP ASSERT].

C. CORRECTNESS OF TYPE SYSTEM

C.1 Preliminary Lemmas

Our type system guarantees that in any well-typed program, all atomic blocks are serializable. We will prove this property, using the following two subject reduction lemmas. The first lemma states the standard property of *preservation*, that is, that evaluation preserves typing:

LEMMA 14 (TYPE SUBJECT REDUCTION). *If $P \vdash \Pi$ and $P \vdash \Pi \rightarrow \Pi'$ then $P \vdash \Pi'$.*

PROOF. See Appendix D. □

The second subject reduction lemma shows that evaluation preserves the invariant that two threads are never in critical sections on the same lock. We introduce a new judgment $ls \vdash_{\text{cs}} e$ to indicate that the lock set ls contains all locks for which e is in a critical section. That is, for all ρ such that $e = \mathcal{E}[\texttt{in-sync}\ \rho\ e']$, the lock ρ is in the set ls.

$$\boxed{ls \vdash_{\text{cs}} e}$$

[CS EXP]
$$\frac{e \text{ does not contain } \texttt{in-sync}}{\emptyset \vdash_{\text{cs}} e}$$

[CS IN-SYNC]
$$\frac{ls \vdash_{\text{cs}} e \qquad \rho \notin ls}{ls \cup \{\rho\} \vdash_{\text{cs}} \texttt{in-sync}\ \rho\ e}$$

[CS NOT IN-SYNC]
$$\frac{ls \vdash_{\text{cs}} e \qquad \mathcal{E} \text{ does not contain } \texttt{in-sync}}{ls \vdash_{\text{cs}} \mathcal{E}[e]}$$

We extend the notion of well-formed critical sections to states with the judgment $\vdash_{\text{cs}} \Pi$. This judgment ensures that a thread is in a critical section `in-sync` $\rho \ldots$ only if it holds the lock ρ in the heap.

$$\boxed{\vdash_{\text{cs}} \Pi}$$

[CS STATE]
$$\frac{\begin{array}{c} ls_i \vdash_{\text{cs}} T_i \qquad \forall i \in 1..|T| \\ ls_i = \{\rho \mid \sigma(\rho) = \{\!|\ldots|\!\}^i_{c_\rho}\} \qquad \forall i \in 1..|T| \end{array}}{\vdash_{\text{cs}} \langle \sigma, T \rangle}$$

This notion of well-formed critical sections is then preserved under evaluation:

LEMMA 15 (MUTUAL EXCLUSION SUBJECT REDUCTION). *If* $\vdash_{\mathrm{cs}} \Pi$ *and* $P \vdash \Pi \rightarrow \Pi'$ *then* $\vdash_{\mathrm{cs}} \Pi'$.

PROOF. See Appendix E. □

The following lemma leverages these two properties to characterize when a field access is guaranteed to be conflict-free; such conflict-free accesses commute with steps from other threads.

LEMMA 16 (CONFLICTING ACCESSES). *Suppose* $P \vdash \langle \sigma, T \rangle$ *and* $\vdash_{\mathrm{cs}} \langle \sigma, T \rangle$ *and* $P \vdash_{\mathrm{inst}}$ `class` c `{` $\ldots t\ fd\ g \ldots$ `}` *and* $P; E \vdash \rho : c$ *and* $P; E \vdash \sigma$. *If* T_k *accesses* $\rho.fd$ *then:*

(1) g *is* `final` *and* $\forall i \neq k$, T_i *does not write* $\rho.fd$*; or*
(2) g *is* `guarded_by` l *and* $\forall i \neq k$, T_i *does not access* $\rho.fd$*; or*
(3) g *is* `write_guarded_by` l *and if* T_k *is in a critical section on* l[`this` $:= \rho$]*, then* $\forall i \neq k$, T_i *does not write* $\rho.fd$*; or*
(4) g *is* `no_guard`.

PROOF. See Appendix E. □

C.2 Reduction

Our proof that atomic blocks are serializable depends on the following reduction theorem, which is inspired by the work of Cohen and Lamport [1998]. For clarity, we express our reduction theorem in terms of an arbitrary transition system. We then demonstrate that the ATOMICJAVA transition relation exhibits the properties necessary to guarantee serializability. Structuring the theorem in this way has allowed us to apply it in other settings as well [Flanagan et al. 2005].

The statement of this reduction theorem requires some additional notation. For any state predicate $X \subseteq \mathit{State}$ and transition relation $Y \subseteq \mathit{State} \times \mathit{State}$, the transition relation X/Y is obtained by restricting Y to pairs whose first component is in X. Similarly, the transition relation $Y \backslash X$ is the restriction of Y to pairs whose second component is in X. The composition $Y \circ Z$ of two transition relations Y and Z is the set of all transitions (p, r) such that there is a state q and transitions $(p, q) \in Y$ and $(q, r) \in Z$. A transition relation Y *right-commutes* with a transition relation Z if $Y \circ Z \subseteq Z \circ Y$, and Y *left-commutes* with Z if $Z \circ Y \subseteq Y \circ Z$.

Recall that each atomic block should consist of a sequence of right movers followed by an atomic action followed by a sequence of left movers. For each thread i, we partition the set of states into four categories:

—$\mathcal{N}_i$: states where thread i is not currently executing an atomic block;
—$\mathcal{R}_i$: states where thread i is executing the "right-mover" part of some atomic block (before the atomic action);
—$\mathcal{L}_i$: states where thread i is executing the "left-mover" part of some atomic block (after the atomic action); or
—$\mathcal{W}_i$: states where thread i has gone wrong.

We also introduce three transition relations over states:

—$\hookrightarrow_i$: the transition relation describing the behavior of each thread i.
—$\hookrightarrow$: the transition relation describing the behavior of interleaving the behaviors of each thread in a program.
—$\hookrightarrow_c$: the transition relation describing the serialized behavior of a program, in which at most one thread may be in an atomic block at any given moment.

The following reduction theorem then shows that if the $\hookrightarrow_i$ transition relation satisfies certain constraints, then each atomic block is serializable, that is, that the standard semantics ($\hookrightarrow$) and the serial semantics ($\hookrightarrow_c$) are essentially equivalent.

We first describe the necessary conditions and the intuition behind their formulation informally, and we then state the Reduction Theorem formally. In essence, the following must be true of the transition relation $\hookrightarrow_i$ and sets $\mathcal{R}_j$, $\mathcal{L}_j$, $\mathcal{W}_j$, and $\mathcal{N}_j$ to ensure serializability:

(1) Each thread i can be in only one of $\mathcal{R}_i$, $\mathcal{L}_i$, $\mathcal{W}_i$. Thus, it cannot, for example, be simultaneously in the left-mover and right-mover part of an atomic block.
(2) A step by thread i cannot cause a transition from $\mathcal{L}_i$ to $\mathcal{R}_i$. This ensures that no thread ever changes from being in the left-mover part of an atomic block to being in the right-mover part.
(3) No steps are possible for thread i once it goes wrong and enters $\mathcal{W}_i$.
(4) Steps by distinct threads i and j are disjoint. In other words, steps by different thread cannot have the same overall effect on program state.
(5) A step taken by thread i while in $\mathcal{R}_i$ (ie., in the right-mover part of an atomic block) right-commutes with steps taken by any other thread j.
(6) A step taken by thread i while in $\mathcal{L}_i$ (ie., in the left-mover part of an atomic block) left-commutes with steps taken by other thread j.
(7) A step taken by thread i cannot change whether any another thread j is in $\mathcal{R}_j$, $\mathcal{L}_j$, $\mathcal{W}_j$, or $\mathcal{N}_j$. Thus, a thread cannot affect which part of an atomic block another thread is currently executing.

If these seven conditions hold then all execution sequences induced by $\hookrightarrow$ are serializable. Specifically, if a program in state p in which no threads are in atomic blocks evaluates to a similar state q, then q can also be reached by a serialized execution, and if a program in state p goes wrong under the normal semantics, it will go wrong under the serialized semantics (provided that any threads in left-mover parts of atomic blocks eventually exit that block).

THEOREM 17 (REDUCTION). *For all i, let $\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ be sets of states, and $\hookrightarrow_i$ be a transition relation. Suppose for all i,*

(1) *$\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ are pairwise disjoint,*
(2) *($\mathcal{L}_i / {\hookrightarrow_i} \backslash \mathcal{R}_i$) is false,*
(3) *$\mathcal{W}_i / {\hookrightarrow_i}$ is false,*

and for all $j \neq i$,

(4) $\hookrightarrow_i$ *and* $\hookrightarrow_j$ *are disjoint,*
(5) $(\hookrightarrow_i \backslash \mathcal{R}_i)$ *right-commutes with* $\hookrightarrow_j$,
(6) $(\mathcal{L}_i / \hookrightarrow_i)$ *left-commutes with* $\hookrightarrow_j$,
(7) *if* $p \hookrightarrow_i q$, *then* $\mathcal{R}_j(p) \Leftrightarrow \mathcal{R}_j(q)$, $\mathcal{L}_j(p) \Leftrightarrow \mathcal{L}_j(q)$, *and* $\mathcal{W}_j(p) \Leftrightarrow \mathcal{W}_j(q)$.

Let $\mathcal{N}_i = \neg(\mathcal{R}_i \vee \mathcal{L}_i)$, $\mathcal{N} = \forall i.\ \mathcal{N}_i$, $\mathcal{W} = \exists i.\ \mathcal{W}_i$, $\hookrightarrow = \exists i.\ \hookrightarrow_i$, *and* $\hookrightarrow_c = \exists i.\ ((\forall j \neq i.\ \mathcal{N}_j)/\hookrightarrow_i)$. *Suppose* $p \in \mathcal{N}$ *and* $p \hookrightarrow^* q$. *Then the following statements are true.*

(1) *If* $q \in \mathcal{N}$, *then* $p \hookrightarrow_c^* q$.
(2) *If* $q \in \mathcal{W}$ *and* $\forall i.\ q \notin \mathcal{L}_i$, *then* $p \hookrightarrow_c^* q'$ *and* $q' \in \mathcal{W}$.

PROOF. See Flanagan and Qadeer [2003c]. □

C.3 ATOMICJAVA Correctness

We now focus on applying the previous reduction theorem to ATOMICJAVA programs. For simplicity, we consider a fixed program P in the remainder of the proof. We use $\Pi_1 \to \Pi_2$ to abbreviate $P \vdash \Pi_1 \to \Pi_2$, and similarly for the other relations on states. We define the atomicity $\alpha(E, e)$ of an expression e to be a if $P; E \vdash e : t \cdot a$. An examination of the type rules shows that α is a well-defined partial function.

Let WT denote the set of well-typed states $\{\langle \sigma, T \rangle \mid P \vdash \langle \sigma, T \rangle\}$. For each thread i, we partition this set of well-typed states into four categories (corresponding to the sets $\mathcal{N}_i$, $\mathcal{R}_i$, $\mathcal{L}_i$ and $\mathcal{W}_i$ above):

$$
\begin{aligned}
N_i &= WT \cap \{\langle \sigma, T \rangle \mid |T| < i \ \vee\ T_i \not\equiv \mathcal{E}[\texttt{in-atomic}\ e]\} \\
W_i &= WT \cap \{\langle \sigma, T \rangle \mid T_i \equiv \textbf{wrong}\} \\
R_i &= WT \cap \{\langle \sigma, T \rangle \mid P; E \vdash \sigma \ \wedge\ T_i \equiv \mathcal{E}[\texttt{in-atomic}\ e] \ \wedge\ Y(\mathcal{E}, \alpha(E, e)) \not\sqsubseteq \texttt{left}\} \\
L_i &= WT \cap \{\langle \sigma, T \rangle \mid P; E \vdash \sigma \ \wedge\ T_i \equiv \mathcal{E}[\texttt{in-atomic}\ e] \ \wedge\ Y(\mathcal{E}, \alpha(E, e)) \sqsubseteq \texttt{left}\}
\end{aligned}
$$

The function $Y(\mathcal{E}, \alpha(E, e))$ is the atomicity of e simplified with respect to the locks held in the evaluation context $\mathcal{E}$. If e has basic atomicity b, that atomicity cannot be simplified further. However, if e has atomicity $l\,?\,a_1 : a_2$ and is being evaluated in a context in which l is held, we can simplify the atomicity of e to $Y(\mathcal{E}, a_1)$, defined as follows:

$$
\begin{aligned}
Y(\mathcal{E}, b) &= b \\
Y(\mathcal{E}, l\,?\,a_1 : a_2) &= \begin{cases} Y(\mathcal{E}, a_1) & \text{if } \mathcal{E} \equiv \mathcal{E}'[\texttt{in-sync}\ l\ \mathcal{E}''] \\ l\,?\,Y(\mathcal{E}, a_1) : Y(\mathcal{E}, a_2) & \text{otherwise} \end{cases}
\end{aligned}
$$

This function enables us to determine whether we are in the left-mover or right-mover part of an atomic block. For example if $\alpha(E, e) = l\,?\,\texttt{mover}:\texttt{atomic}$, and $\mathcal{E} \equiv \texttt{in-sync}\ l\ [\ ]$, then we can consider e to have atomicity

$$
Y(\mathcal{E}, l\,?\,\texttt{mover}:\texttt{atomic}) = Y(\mathcal{E}, \texttt{mover}). = \texttt{mover}
$$

in the current context, making it be in the left-mover part of the atomic block.

In contrast, if $\mathcal{E} \equiv \texttt{in-sync}\ m\ [\ ]$, then e is considered to have atomicity

$$Y(\mathcal{E}, l\,?\,\texttt{mover}:\texttt{atomic}) = l\,?\,Y(\mathcal{E}, \texttt{mover}):Y(\mathcal{E}, \texttt{atomic}) = l\,?\,\texttt{mover}:\texttt{atomic}$$

and is therefore in the right-mover part of the atomic block.

We now have the necessary machinery to prove the fundamental correctness property of our type system. If program execution starts from a well-typed initial state Π_1 and reaches a subsequent state Π_2, where no thread in Π_1 or Π_2 is inside an `in-atomic` block, then Π_2 can also be reached from Π_1 according to the coarser serial transition relation $\mapsto$, where each atomic block is executed "atomically" and is not interleaved with steps from other threads. Also, if Π_2 goes wrong, then provided no thread in Π_2 is in the left-mover part of an atomic block, we can reach some state Π_2' that also goes wrong from Π_1 via the serial transition relation $\mapsto$.

THEOREM 18 (CORRECTNESS). *Let Π_1 be a state such that $P \vdash \Pi_1$ and $\vdash_{\text{cs}} \Pi_1$. Suppose $\forall i.\ N_i(\Pi_1)$ and $\Pi_1 \rightarrow^* \Pi_2$. Then the following statements are true.*

(1) *If $\forall i.\ N_i(\Pi_2)$, then $\Pi_1 \mapsto^* \Pi_2$.*
(2) *If $\exists i.\ W_i(\Pi_2)$ and $\forall i.\ \neg L_i(\Pi_2)$, then $\Pi_1 \mapsto^* \Pi_2'$ and $\exists i.\ W_i(\Pi_2')$.*

PROOF. We show that for all thread indices i, the antecedents of the Reduction Theorem (Theorem 17) are satisfied:

1. R_i, L_i, and W_i are pairwise disjoint,
2. $(L_i/\!\rightarrow_i\!\backslash R_i)$ is false,
3. $W_i/\!\rightarrow_i$ is false,

and for all thread indices $j \neq i$,

4. $\rightarrow_i$ and $\rightarrow_j$ are disjoint,
5. $(\rightarrow_i\!\backslash R_i)$ right-commutes with $\rightarrow_j$,
6. $(L_i/\!\rightarrow_i)$ left-commutes with $\rightarrow_j$,
7. if $\Pi_1 \rightarrow_i \Pi_2$, then $R_j(\Pi_1) \Leftrightarrow R_j(\Pi_2)$, $L_j(\Pi_1) \Leftrightarrow L_j(\Pi_2)$, and $W_j(\Pi_1) \Leftrightarrow W_j(\Pi_2)$.

Then, we obtain the desired result from the Reduction Theorem by substituting

—the set W_i for $\mathcal{W}_i$,
—the set R_i for $\mathcal{R}_i$,
—the set L_i for $\mathcal{L}_i$,
—the relation $\rightarrow_i$ for $\hookrightarrow_i$,
—the relation $\mapsto_i$ for $\hookrightarrow_c$,
—the state Π_1 for p, and
—the state Π_2 for q. □

These seven statements are shown as follows. Their proofs refer to several small helper lemmas listed after the proof. The most important of these additional lemmas is the Sequentiality Lemma (Lemma 20), which shows that the atomicity of evaluating expression $\mathcal{E}[e]$ is greater than the atomicity of evaluating e and then evaluating $\mathcal{E}[v]$, where e has been replaced by the `const` value v. Thus, evaluation never causes the atomicity of an expression to become larger.

(1) By the definition of L_i, R_i, and W_i.
(2) Suppose $\Pi_1 \rightarrow_i \Pi_2$ where $\Pi_1 \in L_i$ and $\Pi_2 \in R_i$. The proof proceeds by a case analysis on the transition rules for $\Pi_1 \rightarrow_i \Pi_2$.
(3) By definition of W_i and inspection of the transition rules for $\rightarrow_i$.
(4) The transition relation $\rightarrow_i$ changes the expression representing thread i but leaves the expressions of all other threads unchanged. Therefore, $\rightarrow_i$ and $\rightarrow_j$ are disjoint for all $i \neq j$.
(5) We show that $(\rightarrow_i \backslash R_i)$ right-commutes with $\rightarrow_j$ as follows. Suppose $\Pi_1 \rightarrow_i \Pi_2 \rightarrow_j \Pi_3$ where $i \neq j$ and $\Pi_2 \in R_i$. We proceed by case analysis on the transition rule for $\Pi_1 \rightarrow_i \Pi_2$:

—[RED NEW]: Suppose the newly created object is ρ. The step from thread j cannot access ρ, because thread j must be well-typed in an environment that does not contain ρ. Thus the two steps access disjoint sets of objects and commute.

—[RED READ]: In this case,

$$\begin{array}{ll} \Pi_1 = \langle \sigma, T \rangle & T_i \equiv \mathcal{E}[\texttt{in-atomic}\ \mathcal{E}'[\rho.fd]] \\ \Pi_2 = \langle \sigma, T' \rangle & T'_i \equiv \mathcal{E}[\texttt{in-atomic}\ \mathcal{E}'[v]] \end{array}$$

where $\sigma(\rho.fd) = v$. We proceed by case analysis on the guard annotation g on the field declaration for fd:

—$g = \texttt{final}$: Lemma 16 indicates that no threads may write to $\rho.fd$, so the two steps commute.

—$g = \texttt{guarded\_by}\ l$: Since $P \vdash \Pi_2$ and $\vdash_{cs} \Pi_2$ by Lemmas 14 and 15, no other thread may access $\rho.fd$ by Lemma 16. Thus the two steps commute since they operate on disjoint sets of shared data.

—$g = \texttt{no\_guard}$: For simplicity, we assume that type of field fd is `int`, although the same reasoning applies for any type. This implies that $\alpha(E, \rho.fd) = \texttt{atomic}$. If E is the environment used to show $P; E \vdash \sigma$, then

$$Y(\mathcal{E}, \alpha(E, \mathcal{E}'[v])) \not\sqsubseteq \texttt{left} \quad \text{since } \Pi_2 \in R_i \quad (\dagger)$$

Also,

$$\begin{array}{ll} \texttt{atomic} & \\ \sqsupseteq Y(\mathcal{E}, \alpha(E, \mathcal{E}'[\rho.fd])) & \text{by Lemma 21} \\ \sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \alpha(E, \rho.fd)); \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 20} \\ \sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \texttt{atomic}); \alpha(E, \mathcal{E}'[v])) & \text{by assumption} \\ \sqsupseteq Y(\mathcal{E}, \texttt{atomic}; \alpha(E, \mathcal{E}'[v])) & \text{by definition of } Y \\ \sqsupseteq \texttt{atomic}; Y(\mathcal{E}, \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 19(3) and def. of } Y \\ \sqsupset \texttt{atomic} & \text{by Lemma 19(1) and } (\dagger) \end{array}$$

However, this is a contradiction.

— g = `write_guarded_by` l: If thread i is in a critical section on l, then Lemma 16 indicates that no other threads are writing to $\rho.fd$, so the two steps commute.
If thread i is not in a critical section on l, then inspection of the type rules indicates that $Y(\mathcal{E}, \alpha(E, \rho.fd)) = \texttt{atomic}$, and we proceed as in the case for `no_guard` to reach a contradiction.

— [RED SYNC]: Let

$$\Pi_1 = \langle \sigma, T \rangle \qquad T_i \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[\texttt{sync } \rho\ e]]$$
$$\Pi_2 = \langle \sigma, T' \rangle \qquad T_i' \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[\texttt{in-sync } \rho\ e]]$$

If thread i acquires the lock on object ρ, then the step by j cannot be a step taken by [RED SYNC], [RED IN-SYNC], or [RED RE-SYNC] where the object being manipulated is ρ. These are the only three steps which could interfere with acquiring lock ρ in thread i. Thus, the two steps commute.

— [RED IN-SYNC]: Let

$$\Pi_1 = \langle \sigma, T \rangle \qquad T_i \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[\texttt{in-sync } \rho\ v]]$$
$$\Pi_2 = \langle \sigma, T' \rangle \qquad T_i' \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[v]]$$

If E is the environment used to show $P; E \vdash \sigma$, then

$$Y(\mathcal{E}, \alpha(E, \mathcal{E}'[v])) \not\sqsubseteq \texttt{left} \quad \text{since } \Pi_2 \in R_i \quad (\dagger)$$

Also,

$$\begin{array}{ll}
\texttt{atomic} & \\
\sqsupseteq Y(\mathcal{E}, \alpha(E, \mathcal{E}'[\texttt{in-sync } \rho\ v])) & \text{by Lemma 21} \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \alpha(E, \texttt{in-sync } \rho\ v)); \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 20} \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \texttt{left}); \alpha(E, \mathcal{E}'[v])) & \text{by rule [EXP IN-SYNC]} \\
\sqsupseteq Y(\mathcal{E}, \texttt{left}; \alpha(E, \mathcal{E}'[v])) & \text{by definition of } Y \\
\sqsupseteq \texttt{left}; Y(\mathcal{E}, \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 19(3) and def. of } Y \\
\sqsupset \texttt{atomic} & \text{by Lemma 19(2) and } (\dagger)
\end{array}$$

However, the last line is a contradiction, so this case cannot happen.

— [RED FORK]: Let

$$\Pi_1 = \langle \sigma, T \rangle \qquad T_i \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[\rho.\texttt{fork}]]$$
$$\Pi_2 = \langle \sigma, T' \rangle \qquad T_i' \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[0]]$$

Assume that E is the environment used to show $P; E \vdash \sigma$. We have

$$Y(\mathcal{E}, \alpha(E, \mathcal{E}'[0])) \not\sqsubseteq \texttt{left} \quad \text{since } \Pi_2 \in R_i \quad (\dagger)$$

Also,

$$\begin{array}{ll}
\texttt{atomic} & \\
\sqsupseteq Y(\mathcal{E}, \alpha(E, \mathcal{E}'[\rho.\texttt{fork}])) & \text{by Lemma 21} \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \alpha(E, \rho.\texttt{fork})); \alpha(E, \mathcal{E}'[0])) & \text{by Lemma 20} \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \texttt{atomic}); \alpha(E, \mathcal{E}'[0])) & \text{by [EXP FORK]}
\end{array}$$

$$\begin{array}{ll} \sqsupseteq Y(\mathcal{E}, \texttt{atomic}; \alpha(E, \mathcal{E}'[0])) & \text{by definition of } Y \\ \sqsupseteq \texttt{atomic}; Y(\mathcal{E}, \alpha(E, \mathcal{E}'[0])) & \text{by Lemma 19(3) and def. of } Y \\ \sqsupset \texttt{atomic} & \text{by Lemma 19(1) and } (\dagger) \end{array}$$

However, the last line is a contradiction.

—[RED INVOKE], [RED LET], [RED IF-TRUE], [RED IF-FALSE], [RED WHILE], [RED RE-SYNC], [RED ATOMIC],
[RED IN-ATOMIC], [RED WRONG]: Trivial, since the store σ does not change.

(6) We show that $(L_i/\rightarrow_i)$ left-commutes with $\rightarrow_j$ as follows. Suppose $\Pi_1 \rightarrow_j \Pi_2 \rightarrow_i \Pi_3$ where $i \neq j$ and $\Pi_2 \in L_i$. We proceed by case analysis on the transition rule for $\Pi_2 \rightarrow_i \Pi_3$:

—[RED NEW]: As above.

—[RED READ]: In this case,

$$\begin{array}{ll} \Pi_2 = \langle \sigma, T \rangle & T_i \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[\rho.fd]] \\ \Pi_3 = \langle \sigma, T' \rangle & T'_i \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[v]] \end{array}$$

where $\sigma(\rho.fd) = v$. We proceed by case analysis on the guard annotation g on the field declaration for fd:

—$g = \texttt{final}$: Lemma 16 indicates that no threads may write to $\rho.fd$, so the two steps commute.

—$g = \texttt{guarded\_by}\ l$: Since $P \vdash \Pi_2$ and $\vdash_{cs} \Pi_2$ by Lemmas 14 and 15, no other thread may access $\rho.fd$ by Lemma 16. Thus the two steps commute since they operate on disjoint sets of shared data.

—$g = \texttt{no\_guard}$: For simplicity, we assume that type of field fd is `int`, although the same reasoning applies for any type. This implies that $\alpha(E, \rho.fd) = \texttt{atomic}$. If E is the environment used to show $P; E \vdash \sigma$, then

$$\begin{array}{ll} \texttt{left} & \\ \sqsupseteq Y(\mathcal{E}, \alpha(E, \mathcal{E}'[\rho.fd])) & \text{since } \Pi_2 \in L_i \\ \sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \alpha(E, \rho.fd)); \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 20, for some } v \\ \sqsupseteq Y(\mathcal{E}, \texttt{atomic}; \alpha(E, \mathcal{E}'[v])) & \text{by assumption and def. of } Y \\ \sqsupseteq Y(\mathcal{E}, \texttt{atomic}); Y(\mathcal{E}, \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 19(3)} \\ \sqsupseteq \texttt{atomic}; Y(\mathcal{E}, \alpha(E, \mathcal{E}'[v])) & \text{by definition of } Y \\ \sqsupseteq \texttt{atomic} & \text{by Theorem 3} \end{array}$$

However, this last line is a contradiction, because $\texttt{left} \not\sqsupseteq \texttt{atomic}$.

—$g = \texttt{write\_guarded\_by}\ l$: If thread i is in a critical section on l, then Lemma 16 indicates that no other threads are writing to $\rho.fd$, so the two steps commute.
If thread i is not in a critical section on l, then inspection of the type rules indicates that $Y(\mathcal{E}, \alpha(E, \rho.fd)) = \texttt{atomic}$, and we proceed as in the case for `no_guard` to reach a contradiction.

—[RED SYNC]:

$$\begin{array}{ll} \Pi_2 = \langle \sigma, T \rangle & T_i \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[\texttt{sync } \rho\ e]] \\ \Pi_3 = \langle \sigma, T' \rangle & T'_i \equiv \mathcal{E}[\texttt{in-atomic } \mathcal{E}'[\texttt{in-sync } \rho\ e]] \end{array}$$

Also, thread i is not in a critical section on ρ. Assume that E is the environment used to show $P; E \vdash \sigma$.

$$
\begin{array}{ll}
\texttt{left} & \\
\sqsupseteq Y(\mathcal{E}, \alpha(E, \mathcal{E}'[\texttt{sync}\ \rho\ e])) & \text{since } \Pi_2 \in L_i \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \alpha(E, \texttt{sync}\ \rho\ e)); \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 20, for some } v \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \alpha(E, \texttt{sync}\ \rho\ e))) & \text{by Theorem 3 and Lemma 19(4)} \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', S(\rho, a))) & \text{by rule [EXP SYNC], for some } a \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \rho\,?\,\texttt{const}:\texttt{atomic})) & \text{by def. of } S \text{ and Lemma 19(4)} \\
= \rho\,?\,\texttt{const}:\texttt{atomic} & \text{since } \rho \text{ not held in } \mathcal{E}' \text{ or } \mathcal{E}
\end{array}
$$

However, $\texttt{left} \not\sqsupseteq \rho\,?\,\texttt{const}:\texttt{atomic}$, and a contradiction exists, so this case cannot happen.

—[RED IN-SYNC]: Let

$$
\begin{array}{ll}
\Pi_2 = \langle \sigma, T \rangle & T_i \equiv \mathcal{E}[\texttt{in-atomic}\ \mathcal{E}'[\texttt{in-sync}\ \rho\ v]] \\
\Pi_3 = \langle \sigma, T' \rangle & T'_i \equiv \mathcal{E}[\texttt{in-atomic}\ \mathcal{E}'[v]]
\end{array}
$$

If thread i releases the lock on object ρ, then the step by j cannot be a step taken by [RED SYNC], [RED IN-SYNC], or [RED RE-SYNC] where the object being manipulated is ρ. These are the only three steps which could interfere with acquiring lock ρ in thread i. Thus, the two steps commute.

—[RED FORK]: Let

$$
\begin{array}{ll}
\Pi_2 = \langle \sigma, T \rangle & T_i \equiv \mathcal{E}[\texttt{in-atomic}\ \mathcal{E}'[\rho.\texttt{fork}]] \\
\Pi_3 = \langle \sigma, T' \rangle & T'_i \equiv \mathcal{E}[\texttt{in-atomic}\ \mathcal{E}'[0]]
\end{array}
$$

Assume that E is the environment used to show $P; E \vdash \sigma$. We have

$$
\begin{array}{ll}
\texttt{left} & \\
\sqsupseteq Y(\mathcal{E}, \alpha(E, \texttt{in-atomic}\ \mathcal{E}'[\rho.\texttt{fork}])) & \text{since } \Pi_2 \in L_i \\
\sqsupseteq Y(\mathcal{E}, Y(\mathcal{E}', \alpha(E, \rho.\texttt{fork})); \alpha(E, \mathcal{E}'[0])) & \text{by Lemma 20, for some } v \\
\sqsupseteq Y(\mathcal{E}, \texttt{atomic}; \alpha(E, \mathcal{E}'[0])) & \text{by [EXP FORK] and def. of } Y \\
\sqsupseteq Y(\mathcal{E}, \texttt{atomic}); Y(\mathcal{E}, \alpha(E, \mathcal{E}'[0])) & \text{by Lemma 19(3)} \\
\sqsupseteq \texttt{atomic}; Y(\mathcal{E}, \alpha(E, \mathcal{E}'[0])) & \text{by definition of } Y \\
\sqsupseteq \texttt{atomic} & \text{by Theorem 3}
\end{array}
$$

However, this last line is a contradiction because $\texttt{left} \not\sqsupseteq \texttt{atomic}$.

—[RED INVOKE], [RED LET], [RED IF-TRUE], [RED IF-FALSE], [RED WHILE], [RED RE-SYNC], [RED ATOMIC],
[RED IN-ATOMIC], [RED WRONG]: As before.

—Suppose $\Pi_1 \rightarrow_i \Pi_2$. If this step is not a fork step, then threads other than i do not change in going from Π_1 to Π_2. Therefore, $R_j(\Pi_1) \Leftrightarrow R_j(\Pi_2)$, $L_j(\Pi_1) \Leftrightarrow L_j(\Pi_2)$, and $W_j(\Pi_1) \Leftrightarrow W_j(\Pi_2)$.
If this step forks a new thread k, then that new thread's expression does not contain `in-atomic` (see rule [RED FORK]). Thus $N_k(\Pi_1)$ and $N_k(\Pi_2)$. Furthermore, since no threads other than i and k change when going from Π_1 to Π_2, we have that $R_j(\Pi_1) \Leftrightarrow R_j(\Pi_2)$, $L_j(\Pi_1) \Leftrightarrow L_j(\Pi_2)$, and $W_j(\Pi_1) \Leftrightarrow W_j(\Pi_2)$ for all $j \neq i$. □

The following supporting lemmas are used in the previous proof. We first state a number of properties regarding atomicities and the function Y. They all follow from definitions or from routine induction over the structure of an atomicity.

LEMMA 19 (ATOMICITY PROPERTIES)

(1) If $a \not\sqsubseteq \texttt{left}$ then $\texttt{atomic}; a \sqsupset \texttt{atomic}$.
(2) If $a \not\sqsubseteq \texttt{left}$ then $\texttt{left}; a \sqsupset \texttt{atomic}$.
(3) $Y(\mathcal{E}, a_1; a_2) \sqsupseteq Y(\mathcal{E}, a_1); Y(\mathcal{E}, a_2)$.
(4) If $a \sqsubseteq a'$ then $Y(\mathcal{E}, a) \sqsubseteq Y(\mathcal{E}, a')$.
(5) $Y(\texttt{in-sync}\ \rho\ \mathcal{E}, a) = IS(\rho, Y(\mathcal{E}, a))$.
(6) If $Y(\mathcal{E}, R(a)) \sqsubseteq \texttt{cmpd}$ then $Y(\mathcal{E}, a) \sqsubseteq \texttt{atomic}$.
(7) For all P, E, a, a', if $P; E \vdash a \uparrow a'$ then $a \sqsubseteq a'$.

We next show that the atomicity of an expression $\mathcal{E}[e]$ is the atomicity of e composed with the atomicity of $\mathcal{E}[v]$, for any v. This lemma relates evaluation order with how atomicities are computed in the type system and shows that evaluation never causes the atomicity of an expression to become larger. (For simplicity, we only show containment in one direction, since that is all that is needed in our proofs.)

LEMMA 20 (SEQUENTIALITY). *For all contexts $\mathcal{E}$, well-formed environments E, and values v, if $\alpha(E, \mathcal{E}[e])$ is defined and $\alpha(E, \mathcal{E}[v])$ is defined and e is not a value, then $Y(\mathcal{E}, \alpha(E, e)); \alpha(E, \mathcal{E}[v]) \sqsubseteq \alpha(E, \mathcal{E}[e])$.*

PROOF. We proceed by induction over $\mathcal{E}$, showing a few representative cases:

—$\mathcal{E} \equiv [\ ]$:

$$\begin{aligned}
\alpha(E, \mathcal{E}[e]) &= \alpha(E, ([])[e]) \\
&= Y([], \alpha(E, e)) \\
&= Y([], \alpha(E, e)); \texttt{const} \\
&= Y([], \alpha(E, e)); \alpha(E, v) \\
&= Y(\mathcal{E}, \alpha(E, e)); \alpha(E, \mathcal{E}[v])
\end{aligned}$$

—$\mathcal{E} \equiv \mathcal{E}'.fd$: Assume that fd is guarded by some lock l. (The other three cases for different guards are similar.) From rule [EXP REF], we know that

$$\begin{aligned}
&P; E \vdash \mathcal{E}'[e] : cn\langle l_{1..n}\rangle \cdot a \\
&P; E \vdash (l\ ?\,\texttt{mover} : \texttt{error})[\texttt{this} := e, x_j := l_j\ ^{j \in 1..n}] \uparrow a'
\end{aligned}$$

Using Lemma 23, we may also conclude that

$$\begin{aligned}
&P; E \vdash (l\ ?\,\texttt{mover} : \texttt{error})[\texttt{this} := v, x_j := l_j\ ^{j \in 1..n}] \uparrow a'' \\
&a'' \sqsubseteq a'
\end{aligned}$$

Then

$$\begin{aligned}
\alpha(E, \mathcal{E}[e]) &= \alpha(E, \mathcal{E}'[e].fd) \\
&= \alpha(E, \mathcal{E}'[e]); a' && \text{from [EXP REF]} \\
&\sqsupseteq Y(\mathcal{E}', \alpha(E, e)); \alpha(E, \mathcal{E}'[v]); a' && \text{by IH} \\
&\sqsupseteq Y(\mathcal{E}', \alpha(E, e)); \alpha(E, \mathcal{E}'[v]); a'' && \text{from above} \\
&= Y(\mathcal{E}', \alpha(E, e)); \alpha(E, \mathcal{E}[v]) && \text{by [EXP REF]} \\
&= Y(\mathcal{E}, \alpha(E, e)); \alpha(E, \mathcal{E}[v])
\end{aligned}$$

The last line uses the fact the $\mathcal{E}$ and $\mathcal{E}'$ contain the same `in-sync` operations. Thus, $Y(\mathcal{E}, a) = Y(\mathcal{E}', a)$ for all atomicities a.

—$\mathcal{E} \equiv$ `let` x `=` $\mathcal{E}'$ `in` e':

$$\begin{array}{ll}
\alpha(E, \mathcal{E}[e]) & \\
= \alpha(E, \texttt{let } x \texttt{ = } \mathcal{E}'[e] \texttt{ in } e') & \\
= \alpha(E, \mathcal{E}'[e]); a & \text{where } \mathcal{E}'[e] \text{ has type } t \text{ and} \\
 & P; E \vdash \alpha((E, t\ x), e')[x := \mathcal{E}'[e]] \uparrow a \\
 & \text{by rule [EXP LET]} \\
\sqsupseteq (Y(\mathcal{E}', \alpha(E, e)); \alpha(E, \mathcal{E}'[v])); a & \text{by IH} \\
\sqsupseteq (Y(\mathcal{E}', \alpha(E, e)); \alpha(E, \mathcal{E}'[v])); a' & \text{where } P; E \vdash \alpha((E, t\ x), e')[x := \mathcal{E}'[v]] \uparrow \\
 & a' \text{ by Lemma 23} \\
= Y(\mathcal{E}', \alpha(E, e)); \alpha(E, \mathcal{E}[v]) & \text{by rule [EXP LET]} \\
= Y(\mathcal{E}, \alpha(E, e)); \alpha(E, \mathcal{E}[v]) & \text{as in the previous case}
\end{array}$$

—$\mathcal{E} \equiv$ `sync` $\mathcal{E}'\ e'$: Cannot happen, since $\mathcal{E}'[e]$ must be a constant value and e is not a value.

—$\mathcal{E} \equiv$ `in-sync` $\rho\ \mathcal{E}'$:

$$\begin{array}{ll}
\alpha(E, \mathcal{E}[e]) & \\
= \alpha(E, \texttt{in-sync } \rho\ \mathcal{E}'[e]) & \\
= IS(\rho, \alpha(E, \mathcal{E}'[e])) & \text{by rule [EXP IN-SYNC]} \\
\sqsupseteq IS(\rho, Y(\mathcal{E}', \alpha(E, e)); \alpha(E, \mathcal{E}'[v])) & \text{by IH} \\
= IS(\rho, Y(\mathcal{E}', \alpha(E, e))); IS(\rho, \alpha(E, \mathcal{E}'[v])) & \text{by dist. of } IS \text{ over ;} \\
= Y(\texttt{in-sync } \rho\ \mathcal{E}', \alpha(E, e)); IS(\rho, \alpha(E, \mathcal{E}'[v])) & \text{by Lemma 19(5)} \\
= Y(\mathcal{E}, \alpha(E, e)); IS(\rho, \alpha(E, \mathcal{E}'[v])) & \text{by def. of } \mathcal{E} \\
= Y(\mathcal{E}, \alpha(E, e)); \alpha(E, \mathcal{E}[v]) & \text{by rule [EXP IN-SYNC]} \quad \square
\end{array}$$

Finally, we show that an expression `in-atomic` e is only evaluated when the locks held by the current thread allow e's atomicity to be simplified to `atomic`.

LEMMA 21 (IN-ATOMIC BLOCKS ARE ATOMIC). *If* $\alpha(E, \mathcal{E}[\texttt{in-atomic } e]) \sqsubseteq \texttt{cmpd}$ *then* $Y(\mathcal{E}, \alpha(E, e)) \sqsubseteq \texttt{atomic}$.

PROOF. We begin by first applying the Sequentiality Lemma to focus in on the atomicity of the expression `in-atomic` e embedded inside the evaluation context $\mathcal{E}$. Once we have expressed the overall atomicity in terms of this atomicity, we can proceed to show that the atomicity must be no larger than `atomic`.

$$\begin{array}{lll}
\texttt{cmpd} & \sqsupseteq \alpha(E, \mathcal{E}[\texttt{in-atomic } e]) & \\
 & \sqsupseteq Y(\mathcal{E}, \alpha(E, \texttt{in-atomic } e)); \alpha(E, \mathcal{E}[v]) & \text{by Lemma 20, where } v \text{ is 0 or } \texttt{null} \\
 & \sqsupseteq Y(\mathcal{E}, R(\alpha(E, e))); \alpha(E, \mathcal{E}[v]) & \text{by [EXP IN-ATOMIC]} \\
 & \sqsupseteq Y(\mathcal{E}, R(\alpha(E, e))) & \text{by Theorem 3}
\end{array}$$

Lemma 19(6) then indicates that $Y(\mathcal{E}, \alpha(E, e)) \sqsubseteq \texttt{atomic}$. $\square$

D. TYPE SUBJECT REDUCTION

In this section, we prove that types and atomicities are preserved under evaluation. The preliminary lemmas are routine, and much of their structure is derived directly from previous work on similar systems [Flatt et al. 1998; Abadi et al. 2006]. Therefore, we primarily focus on the novel aspects of the language, including how atomicities and dependent types are handled.

We start by showing that all operations on atomicities are monotonic.

LEMMA 22 (ATOMICITY MONOTONICITY)

(1) If $a_1 \sqsubseteq a_2$, then for all a:

$$\begin{array}{rclcrclcrcl} a_1;a & \sqsubseteq & a_2;a & \quad & a \sqcup a_1 & \sqsubseteq & a \sqcup a_2 & \quad & S(l,a_1) & \sqsubseteq & S(l,a_2) \\ a;a_1 & \sqsubseteq & a;a_2 & & a_1^* & \sqsubseteq & a_2^* & & R(a_1) & \sqsubseteq & R(a_2) \\ a_1 \sqcup a & \sqsubseteq & a_2 \sqcup a & & \theta(a_1) & \sqsubseteq & \theta(a_2) & & IS(l,a_1) & \sqsubseteq & IS(l,a_2) \end{array}$$

(2) If $P;E \vdash a_1 \uparrow a_1'$ and $P;E \vdash a_2 \uparrow a_2'$ then $a_1' \sqsubseteq a_2'$.

PROOF. Follows from the definitions of these operations. □

The following technical lemma shows a subtle, but important property of lifting used in subsequent proofs.

LEMMA 23 (LIFTING AFTER SUBSTITUTION). *If $P;E \vdash a[x := e'] \uparrow a'$ and $P;E \vdash a[x := e''] \uparrow a''$ and e' is not a value, then $a'' \sqsubseteq a'$.*

PROOF. Proof is by induction on a:

—$a = b$: In this case, $a'' = (b[x := e'']) = b = (b[x := e']) = a'$.

—$a = l\,?\,a_1 : a_2$: By induction, the atomicities of the subterms a_1 and a_2 are related as follows, for $i \in 1..2$:

$$\begin{array}{l} P;E \vdash a_i[x := e'] \uparrow a_i' \\ P;E \vdash a_i[x := e''] \uparrow a_i'' \\ a_i'' \sqsubseteq a_i' \end{array}$$

We consider two cases:

—x is free in l: Thus, $l[x := e']$ is not a value and $P;E \nvdash_{\text{lock}} l[x := e']$. Therefore, $a' = a_1' \sqcup a_2'$.
If $l[x := e'']$ is a value, then $a'' = (l[x := e'']\,?\,a_1'' : a_2'') \sqsubseteq a'$.
If $l[x := e'']$ is not a value, then $a'' = (a_1'' \sqcup a_2'') \sqsubseteq a'$.

—x is not free in l: In this case, $l[x := e'] = l = l[x := e'']$.
If $P;E \vdash_{\text{lock}} l$ then $a'' = (l\,?\,a_1'' : a_2'') \sqsubseteq (l\,?\,a_1' : a_2') = a'$.
If $P;E \nvdash_{\text{lock}} l$ then $a'' = (a_1'' \sqcup a_2'') \sqsubseteq (a_1' \sqcup a_2') = a'$ □

The following context lemma states that if an expression $\mathcal{E}[e]$ is well-typed, then so is e.

LEMMA 24 (CONTEXT SUBEXPRESSION). *Suppose there is a deduction that concludes $P;E \vdash \mathcal{E}[e] : t \cdot a$. Then that deduction contains, at a position corresponding to the hole in $\mathcal{E}$, a subdeduction that concludes $P;E \vdash e : t' \cdot a'$.*

PROOF. By induction over the derivation of $P;E \vdash \mathcal{E}[e] : t \cdot a$. □

The next lemma shows that a subexpression e_1 may be replaced by a different subexpression with the same type. Moreover, if the subexpression's atomicity (nonstrictly) decreases, then so does the atomicity of the whole expression. The lemma requires that e_1 is not a value, in which case e_1 will not appear in the inferred types or atomicities.

LEMMA 25 (CONTEXT REPLACEMENT). *Suppose a deduction concluding* $P; E \vdash \mathcal{E}[e_1] : t \cdot a$ *contains a deduction concluding* $P; E \vdash e_1 : t' \cdot a_1$ *at a position corresponding to the hole in* $\mathcal{E}$. *If* e_1 *is not a value and* $P; E \vdash e_2 : t' \cdot a_2$ *and* $a_2 \sqsubseteq a_1$ *then* $P; E \vdash \mathcal{E}[e_2] : t \cdot a'$ *where* $a' \sqsubseteq a$.

PROOF. Proof is by induction on the structure of $\mathcal{E}$. We consider two representative cases:

—$\mathcal{E} \equiv \texttt{if}\ \mathcal{E}'\ f_2\ f_3$: Since $P; E \vdash \mathcal{E}[e_1] : t \cdot a$ is derivable only by rule [EXP IF], it must be that:

$$
\begin{array}{l}
P; E \vdash \mathcal{E}'[e_1] : \texttt{int} \cdot \hat{a}_1 \\
P; E \vdash f_i : t \cdot \hat{a}_i \qquad i \in 2..3 \\
a = \hat{a}_1; (\hat{a}_2 \sqcup \hat{a}_3)
\end{array}
$$

By the inductive hypothesis, $P; E \vdash \mathcal{E}'[e_2] : \texttt{int} \cdot \hat{a}'_1$ where $\hat{a}'_1 \sqsubseteq \hat{a}_1$. Lemma 22 indicates that

$$a' = \hat{a}'_1; (\hat{a}_2 \sqcup \hat{a}_3) \sqsubseteq \hat{a}_1; (\hat{a}_2 \sqcup \hat{a}_3) = a$$

Rule [EXP IF] thus allows us to conclude $P; E \vdash \mathcal{E}[e_2] : t \cdot a'$ where $a' \sqsubseteq a$.

—$\mathcal{E} \equiv \mathcal{E}'.fd$: We consider only the case where the accessed field is guarded by a lock l. The three other cases are similar. Since $P; E \vdash \mathcal{E}[e_1] : t \cdot a$ is derivable only by rule [EXP REF], it must be that:

$$
\begin{array}{l}
P; E \vdash \mathcal{E}'[e_1] : cn\langle l_{1..n}\rangle \cdot \hat{a}_1 \\
\texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots \hat{t}\ fd\ \texttt{guarded\_by}\ l\ \ldots\ \} \in P \\
\theta = [\texttt{this} := \mathcal{E}'[e_1], x_j := l_j\ ^{j \in 1..n}] \\
P; E \vdash \theta(\hat{t}) \\
P; E \vdash \theta(l)\, ?\, \texttt{mover} : \texttt{error} \uparrow \hat{a}_f \\
t = \theta(\hat{t}) \\
a = \hat{a}_1; \hat{a}_f
\end{array}
$$

By the inductive hypothesis, $P; E \vdash \mathcal{E}'[e_2] : cn\langle l_{1..n}\rangle \cdot \hat{a}_2$ where $\hat{a}_2 \sqsubseteq \hat{a}_1$. Since e_1 is not a value, $\mathcal{E}'[e_1]$ is not a value. Let $\theta' = [\texttt{this} := \mathcal{E}'[e_2], x_j := l_j\ ^{j \in 1..n}]$. Since well-formed types can only contain values, it must be that `this` is not free in t. Thus, $\theta(\hat{t}) = \theta'(\hat{t})$ and $P; E \vdash \theta'(\hat{t})$. Similarly, we can show

$$P; E \vdash \theta'(l)\, ?\, \texttt{mover} : \texttt{error} \uparrow \hat{a}'_f$$

where $\hat{a}'_f \sqsubseteq \hat{a}_f$ with Lemma 23. Using rule [EXP REF], we can conclude that $P; E \vdash \mathcal{E}[e_2] : t \cdot a'$, where $a' = \hat{a}_2; \hat{a}'_f$. Lemma 22 permits us to conclude that $a' = (\hat{a}_2; \hat{a}'_f) \sqsubseteq (\hat{a}_1; \hat{a}_f) = a$. □

Environments can be strengthened with additional variable declarations, as follows.

LEMMA 26 (ENVIRONMENT STRENGTHENING). *Suppose* $E = E', t\ x, E''$ *or* $E = E', \texttt{ghost}\ x, E''$. *If* $P \vdash E$ *then:*

(1) *If* $P; (E', E'') \vdash meth$ *then* $P; E \vdash meth$.
(2) *If* $P; (E', E'') \vdash field$ *then* $P; E \vdash field$.
(3) *If* $P; (E', E'') \vdash a$ *then* $P; E \vdash a$.
(4) *If* $P; (E', E'') \vdash t$ *then* $P; E \vdash t$.
(5) *If* $P; (E', E'') \vdash_{\text{lock}} l$ *then* $P; E \vdash_{\text{lock}} l$.
(6) *If* $P; (E', E'') \vdash e : t \cdot a$ *then* $P; E \vdash e : t \cdot a$.
(7) *If* $P; (E', E'') \vdash a \uparrow a'$ *then* $P; E \vdash a \uparrow a'$.
(8) *If* $P; (E', E''); \rho \vdash \{\!|\, db\,|\!\}^o_c$ *then* $P; E; \rho \vdash \{\!|\, db\,|\!\}^o_c$.

PROOF. By simultaneous induction on all parts of the lemma. □

Judgments from the formal system are preserved under capture-free variable substitution.

LEMMA 27 (SUBSTITUTION). *If* $P; E \vdash v : s \cdot \texttt{const}$ *then:*

(1) *If* $P \vdash (E, s\ x, E')$ *then* $P \vdash (E, E'[x := v])$.
(2) *If* $P; (E, s\ x, E') \vdash meth$ *then* $P; (E, E'[x := v]) \vdash meth[x := v]$.
(3) *If* $P; (E, s\ x, E') \vdash field$ *then* $P; (E, E'[x := v]) \vdash field[x := v]$.
(4) *If* $P; (E, s\ x, E') \vdash a$ *then* $P; (E, E'[x := v]) \vdash a[x := v]$.
(5) *If* $P; (E, s\ x, E') \vdash t$ *then* $P; (E, E'[x := v]) \vdash t[x := v]$.
(6) *If* $P; (E, s\ x, E') \vdash_{\text{lock}} l$ *then* $P; (E, E'[x := v]) \vdash_{\text{lock}} l[x := v]$.
(7) *If* $P; (E, s\ x, E') \vdash e : t \cdot a$ *then* $P; (E, E'[x := v]) \vdash e[x := v] : t[x := v] \cdot a[x := v]$.
(8) *If* $P; (E, s\ x, E') \vdash a \uparrow a'$ *then* $P; (E, E'[x := v]) \vdash (a[x := v]) \uparrow (a'[x := v])$.

PROOF. The proof is by a simultaneous induction on all parts of the lemma. We present details for representative cases in the expression type judgment. In particular, we show that if $P; E \vdash v : s \cdot \texttt{const}$ and $P; (E, s\ x, E') \vdash e : t \cdot a$ then $P; (E, E'[x := v]) \vdash e[x := v] : t[x := v] \cdot a[x := v]$ by induction over the derivation of $P; (E, s\ x, E') \vdash e : t \cdot a$. We consider several representative cases:

—[EXP REF]: Let e be the expression $e'.fd$. We consider only the case where the accessed field is guarded by a lock l. The three other cases are similar. From rule [EXP REF],

$$
\begin{array}{l}
P; (E, s\ x, E') \vdash e' : cn\langle l_{1..n}\rangle \cdot a_e \\
\texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots t'\ fd\ \texttt{guarded\_by}\ l\ \ldots\ \} \in P \\
\theta = [\texttt{this} := e', x_j := l_j\ ^{j \in 1..n}] \\
P; (E, s\ x, E') \vdash \theta(t') \\
P; (E, s\ x, E') \vdash (\theta(l)\,?\,\texttt{mover} : \texttt{error}) \uparrow a_f \\
a = (a_1; a_f) \\
t = \theta(t')
\end{array}
$$

By the induction hypothesis:

$$P;(E, E'[x := v]) \vdash e'[x := v] : cn\langle l_{1..n}\rangle[x := v] \cdot a_e[x := v]$$
$$P;(E, E'[x := v]) \vdash (\theta(t'))[x := v]$$
$$P;(E, E'[x := v]) \vdash (\theta(l)\,?\,\texttt{mover}:\texttt{error})[x := v] \uparrow a_f[x := v]$$

Let $\theta' = [\texttt{this} := e'[x := v], x_j := l_j[x := v]\ ^{j\in 1..n}]$. We next show that $(\theta(t'))[x := v] = \theta'(t')$. There are two cases:

—$x = \texttt{this}$ or $x = x_i$ for some i: Since occurrences of x in t' will already have been replaced by θ, we know that

$$(\theta(t'))[x := v] = t'[\texttt{this} := e'[x := v], x_j := l_j[x := v]\ ^{j\in 1..n}] = \theta'(t')$$

—$x \neq \texttt{this}$ and $x \neq x_i$: The only variables in scope where t' appears are $\texttt{this}$ and $x_{1..n}$. Therefore,

$$\begin{aligned}(\theta(t'))[x := v] &= t'[\texttt{this} := e'[x := v], x_j := l_j[x := v]\ ^{j\in 1..n}, x := v]\\ &= t'[\texttt{this} := e'[x := v], x_j := l_j[x := v]\ ^{j\in 1..n}]\\ &= \theta'(t')\end{aligned}$$

Similarly, $(\theta(l)\,?\,\texttt{mover}:\texttt{error})[x := v] = (\theta'(l)\,?\,\texttt{mover}:\texttt{error})$. From these, it follows that

$$P;(E, E'[x := v]) \vdash (\theta'(t'))$$
$$P;(E, E'[x := v]) \vdash (\theta'(l)\,?\,\texttt{mover}:\texttt{error}) \uparrow a_f[x := v]$$

Therefore, $P;(E, E'[x := v]) \vdash (e'.fd)[x := v] : t[x := v] \cdot (a_e; a_f)[x := v]$ by rule [EXP REF].

—[EXP LET]: Let e be the expression $\texttt{let}\ y\ \texttt{=}\ e_1\ \texttt{in}\ e_2$. It must be that:

$$P;(E, s\ x, E') \vdash e_1 : t_1 \cdot a_1$$
$$P;(E, s\ x, E', t_1\ y) \vdash e_2 : t_2 \cdot a_2$$
$$P;(E, s\ x, E') \vdash t_2[y := e_1]$$
$$P;(E, s\ x, E') \vdash a_2[y := e_1] \uparrow a_2'$$
$$t = t_2[y := e_1]$$
$$a = a_1; a_2'$$

We know that the variable x is different from y because $E, s\ x, E', t_1\ y$ is a well-formed environment. By the induction hypothesis:

$$P;(E, E'[x := v]) \vdash e_1[x := v] : t_1[x := v] \cdot a_1[x := v]$$
$$P;(E, E'[x := v], t_1[x := v]\ y) \vdash e_2[x := v] : t_2[x := v] \cdot a_2[x := v]$$
$$P;(E, E'[x := v]) \vdash (t_2[y := e_1])[x := v]$$
$$P;(E, E'[x := v]) \vdash (a_2[y := e_1])[x := v] \uparrow a_2'[x := v]$$

Since x and y are distinct and y is not free in v,

$$t[x := v] = (t_2[y := e_1])[x := v] = (t_2[x := v])[y := (e_1[x := v])]$$

Similarly, $a_2[x := v] = (a_2[x := v])[y := (e_1[x := v])]$. Therefore,

$$P;(E, E'[x := v]) \vdash (t_2[x := v])[y := (e_1[x := v])]$$
$$P;(E, E'[x := v]) \vdash (a_2[x := v])[y := (e_1[x := v])] \uparrow a_2'[x := v]$$

We may thus conclude $P;(E, E'[x := v]) \vdash e[x := v] : t[x := v] \cdot a[x := v]$ using rule [EXP LET].

—[EXP NEW]: Let e be the expression $\texttt{new}_y\ cn\langle l_{1..n}\rangle(e_{1..k})$. It must be that:

$$\begin{array}{l}\theta = [x_j := l_j\ ^{j\in 1..n}, \texttt{this} := y]\\ P;(E, s\ x, E', \texttt{ghost}\ y) \vdash e_i : \theta(t_i)\cdot a_i \qquad \forall i \in 1..k\\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \mathit{field}_{1..k}\ \mathit{meth}_{1..m}\ \} \in P\\ \mathit{field}_i = t_i\ \mathit{fd}_i\ g_i \qquad \forall i \in 1..k\\ P;(E, s\ x, E') \vdash cn\langle l_{1..n}\rangle\\ t = cn\langle l_{1..n}\rangle\\ a = a_1;\cdots;a_k\end{array}$$

By the induction hypothesis:

$$\begin{array}{l}P;(E, E'[x := v], \texttt{ghost}\ y) \vdash e_i[x := v] : (\theta(t_i))[x := v]\cdot a_i[x := v] \quad \forall i \in 1..k\\ P;E \vdash cn\langle l_{1..n}[x := v]\rangle\end{array}$$

We know that the variable x is different from y because $E, s\ x, E', \texttt{ghost}\ y$ is a well-formed environment, and we can assume x is different from $x_{1..n}$, α-renaming the ghost variables as necessary. This implies that x is not free in t_i, since the only names in scope at the field declarations are this and the ghost variables. Letting $\theta' = [x_j := (l_j[x := v])\ ^{j\in 1..n}, \texttt{this} := y]$, we have

$$(\theta(t_i))[x := v] \ = \ (t_i[x := v])[x_j := (l_j[x := v])\ ^{j\in 1..n}, \texttt{this} := y] \ = \ \theta'(t_i).$$

This permits us to conclude that

$$P;(E, E'[x := v], \texttt{ghost}\ y) \vdash e_i[x := v] : \theta'(t_i)\cdot a_i[x := v] \qquad \forall i \in 1..k$$

We may then use the rule [EXP NEW] to conclude that

$$P;(E, E'[x := v] \vdash \texttt{new}_y\ cn\langle l_{1..n}[x := v]\rangle(e_{1..k}[x := v]) : cn\langle l_{1..n}[x := v]\rangle\cdot a'$$

where $a' = ((a_1;\cdots;a_k)[x := v])$. □

A similar lemma is used to substitute values for ghost variables.

LEMMA 28 (GHOST SUBSTITUTION). *If $P;E \vdash_{\text{lock}} v$, then:*

(1) *If* $P \vdash (E, \texttt{ghost}\ x, E')$ *then* $P \vdash (E, E'[x := v])$.
(2) *If* $P;(E, \texttt{ghost}\ x, E') \vdash \mathit{meth}$ *then* $P;(E, E'[x := v]) \vdash \mathit{meth}[x := v]$.
(3) *If* $P;(E, \texttt{ghost}\ x, E') \vdash \mathit{field}$ *then* $P;(E, E'[x := v]) \vdash \mathit{field}[x := v]$.
(4) *If* $P;(E, \texttt{ghost}\ x, E') \vdash a$ *then* $P;(E, E'[x := v]) \vdash a[x := v]$.
(5) *If* $P;(E, \texttt{ghost}\ x, E') \vdash t$ *then* $P;(E, E'[x := v]) \vdash t[x := v]$.
(6) *If* $P;(E, \texttt{ghost}\ x, E') \vdash_{\text{lock}} l$ *then* $P;(E, E'[x := v]) \vdash_{\text{lock}} l[x := v]$.
(7) *If* $P;(E, \texttt{ghost}\ x, E') \vdash e : t\cdot a$ *then* $P;(E, E'[x := v]) \vdash e[x := v] : t[x := v]\cdot a[x := v]$.
(8) *If* $P;(E, \texttt{ghost}\ x, E') \vdash a \uparrow a'$ *then* $P;(E, E'[x := v]) \vdash (a[x := v]) \uparrow (a'[x := v])$.

PROOF. Proof is by simultaneous induction on all parts, as in the previous lemma. □

When now show that typing is preserved under the evaluation relation $\rightarrow_i$.

RESTATEMENT OF LEMMA 14 (TYPE SUBJECT REDUCTION). *If* $P \vdash \Pi$ *and* $P \vdash \Pi \rightarrow \Pi'$ *then* $P \vdash \Pi'$

PROOF. Suppose that $P \vdash \Pi \rightarrow_i \Pi'$ and let

$$\begin{array}{l} \Pi = \langle \sigma, T \rangle \\ \Pi' = \langle \sigma', T' \rangle \end{array}$$

Since $P \vdash \Pi$, rule [STATE] indicates that:

$$\begin{array}{l} P; E \vdash \sigma \\ P; E \vdash T_i : t_i \cdot a_i \qquad \forall i \in 1..|T| \\ a_i \sqsubseteq \texttt{cmpd} \qquad \forall i \in 1..|T| \end{array}$$

We must show the following:

$$\begin{array}{l} P; E' \vdash \sigma' \\ P; E' \vdash T_i' : t_i \cdot a_i' \qquad \forall i \in 1..|T'| \\ a_i' \sqsubseteq \texttt{cmpd} \qquad \forall i \in 1..|T'| \end{array}$$

We proceed by case analysis on the reduction rule used to take a step, showing several representative cases:

—[RED LET]: In this case,

$$\begin{array}{l} T_k \equiv \mathcal{E}[\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e] \\ T_k' \equiv \mathcal{E}[e[x := v]] \end{array}$$

We show below that $P; E \vdash T_k' : t_k \cdot a_k'$ where $a_k' \sqsubseteq a_k \sqsubseteq \texttt{cmpd}$. Since all other threads and store σ (and hence E) do not change, rule [STATE] yields the desired result.
By Lemma 24, it must be that $P; E \vdash \texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e : s \cdot a_{\text{let}}$. This can only be concluded by rule [EXP LET], which means that

$$\begin{array}{l} P; E \vdash v : t_v \cdot a_v \\ P; E, t_v\ x \vdash e : t_e \cdot a_e \\ P; E \vdash t_e[x := v] \\ P; E \vdash a_e[x := v] \uparrow a_e' \\ a_{\text{let}} = (a_v; a_e') \\ s = (t_e[x := v]) \end{array}$$

By Lemma 27, we know that $P; E \vdash e[x := v] : s \cdot a_{\text{let}}[x := v]$. Note that $a_{\text{let}}[x := v] = a_{\text{let}}$ since x does not appear free in a_{let}. Thus, $P; E \vdash e[x := v] : s \cdot a_{\text{let}}$, and by Lemma 25 we have that $P; E \vdash T_k' : t_k \cdot a_k'$ where $a_k' \sqsubseteq a_k \sqsubseteq \texttt{cmpd}$.

—[RED READ]: In this case,

$$\begin{array}{r} T_k \equiv \mathcal{E}[\rho.fd] \\ T_k' \equiv \mathcal{E}[v] \\ \sigma(\rho) = \{\!|\ \ldots, fd = v, \ldots |\!\}_c^o \end{array}$$

As above, it suffices to show that $P; E \vdash T'_k : t_k \cdot a'_k$ where $a'_k \sqsubseteq a_k \sqsubseteq \texttt{cmpd}$. By Lemma 24, it must be that $P; E \vdash \rho.fd : s \cdot a_{\text{acc}}$. This can only be concluded by rule [EXP REF], which means that

$$\begin{array}{l}
P; E \vdash \rho : cn\langle l_{1..n}\rangle \cdot a' \\
\texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \ \ldots t\ fd\ \texttt{guarded_by}\ l\ \ldots\ \} \in P \\
\theta = [\texttt{this} := \rho, x_j := l_j\ {}^{j\in 1..n}] \\
P; E \vdash \theta(t) \\
P; E \vdash (\theta(l)\,?\,\texttt{mover} : \texttt{error}) \uparrow a'' \\
s = \theta(t) \\
a_{\text{acc}} = (a'; a'')
\end{array}$$

(We assume the field fd is guarded by l; the other cases are similar.) Given that $P; E \vdash \sigma$, it must be that $c = cn\langle l_{1..n}\rangle$. Moreover,

$$P; E; \rho \vdash \{\!|\ \ldots, fd = v, \ldots\ |\!\}^o_c$$

which requires that

$$\begin{array}{l}
P \vdash_{\text{inst}} \texttt{class}\ c\ \{\ \ \ldots t[x_j := l_j\ {}^{j\in 1..n}]\ fd_i\ \texttt{guarded_by}\ l[x_j := l_j\ {}^{j\in 1..n}] \ldots\ \} \\
P; E \vdash v : t[x_j := l_j\ {}^{j\in 1..n}][\texttt{this} := \rho] \cdot \texttt{const}
\end{array}$$

Since this is not free in any l_j and is distinct from all x_j (renaming as necessary), $s = \theta(t) = t[x_j := l_j\ {}^{j\in 1..n}][\texttt{this} := \rho]$. Since $\texttt{const} \sqsubseteq a_{\text{acc}}$, Lemma 25 indicates that $P; E \vdash T'_k : t_k \cdot a'_k$ where $a'_k \sqsubseteq a_k \sqsubseteq \texttt{cmpd}$.

—[RED NEW]: In this case,

$$\begin{array}{l}
T_k \equiv \mathcal{E}[\texttt{new}_y\ c(v_{1..n})] \\
T'_k \equiv \mathcal{E}[\rho] \\
\ \rho \notin dom(\sigma) \\
\sigma' = \sigma[\rho \mapsto \{\!|\ fd_i = v_i\ {}^{i\in 1..n}\ |\!\}^{\perp}_c] \\
E' = (E, c\ p)
\end{array}$$

By Lemma 26, we have that

$$\begin{array}{l}
P; E' \vdash T'_i : t_i \cdot a'_i \text{ and } a'_i \sqsubseteq \texttt{cmpd} \qquad \forall i \neq k \\
P; E'; \rho' \vdash \sigma'(\rho') \qquad \forall \rho' \in dom(\sigma) \\
P; E' \vdash \rho : c \cdot \texttt{const}
\end{array}$$

The last statement enables us to use Lemma 25 to conclude that $P; E \vdash T'_k : t_k \cdot a'_k$ where $a'_k \sqsubseteq a_k \sqsubseteq \texttt{cmpd}$. Once we have shown that $P; E'; \rho \vdash \{\!|\ fd_i = v_i\ {}^{i\in 1..n}\ |\!\}^{\perp}_c$, we may use rule [STATE] to show that the lemma holds in this case. By rule [EXP NEW]:

$$\begin{array}{l}
P; (E, \texttt{ghost}\ y) \vdash v_i : (t_i[x_j := l_j\ {}^{j\in 1..r}, \texttt{this} := y]) \cdot \texttt{const} \qquad \forall i \in 1..n \\
\texttt{class}\ cn\langle \texttt{ghost}\ x_{1..r}\rangle\ \{\ field_{1..m}\ \ldots\ \} \in P \\
field_i = t_i\ fd_i\ g_i \qquad \forall i \in 1..k \\
P; E \vdash cn\langle l_{1..r}\rangle
\end{array}$$

Also,

$$\begin{array}{l}
P \vdash_{\text{inst}} \texttt{class}\ cn\langle l_{1..r}\rangle\ \{\ field'_{1..m}\ \ldots\ \} \\
field'_i = t_i[x_j := l_j^{\,j\in 1..r}]\ fd_i\ g_i[x_j := l_j^{\,j\in 1..r}] \qquad \forall i \in 1..m
\end{array}$$

Lemma 27 permits us to conclude that

$$P; E \vdash v_i[y := \rho] : (t_i[x_j := l_j\ ^{j \in 1..r}, \texttt{this} := y])[y := \rho] \cdot \texttt{const} \qquad \forall i \in 1..n$$

Note that y does not appear free in any l_j and is distinct from this and $x_{1..r}$, and also that values cannot contain ghost variables. Thus, we have

$$P; E \vdash v_i : t_i[x_j := l_j\ ^{j \in 1..r}, \texttt{this} := \rho] \cdot \texttt{const} \qquad \forall i \in 1..n$$

which then allows to conclude that $P; E'; \rho \vdash \{\!| fd_i = v_i\ ^{i \in 1..n} |\!\}_c^{\perp}$ by rule [OBJECT].

—[RED FORK]: In this case,

$$\begin{aligned} T_k &\equiv \mathcal{E}[\rho.\texttt{fork}] \\ T'_k &\equiv \mathcal{E}[0] \\ T'_n &\equiv (\texttt{let}\ x\ \texttt{= new Object() in sync}\ x\ (\rho.\texttt{run}\langle x\rangle\texttt{()})) \end{aligned}$$

where $n = |T|+1$. Since $P; E \vdash \rho.\texttt{fork} : \texttt{int}\cdot\texttt{atomic}$ and $P; E \vdash 0 : \texttt{int}\cdot\texttt{const}$, Lemma 25 concludes that $P; E \vdash T'_k : t_k \cdot a'_k$ where $a'_k \sqsubseteq a_k \sqsubseteq \texttt{cmpd}$.

All other threads in T and store σ (and hence E) do not change, so once we have shown that $P; E \vdash T'_n : t_n \cdot a'_n$ and $a'_n \sqsubseteq \texttt{cmpd}$ we may use rule [STATE] to conclude that the lemma holds in this case.

According to rule [EXP FORK], it must be that

$$\begin{aligned} &P; E \vdash \rho : cn\langle l_{1..n}\rangle \cdot \texttt{const} \\ &\texttt{class}\ cn\langle\texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots\ a'\ \texttt{int run}\langle\texttt{ghost tll}\rangle\texttt{()}\ \{\ e'\ \}\ \ldots\ \} \in P \\ &a' \sqsubseteq (\texttt{tll}\,?\,\texttt{cmpd} : \texttt{error}) \end{aligned}$$

Using the type rules, we can conclude that

$$P; E \vdash T'_n : \texttt{int} \cdot a'_n$$

where $P; E \vdash S(x, a'[\texttt{tll} := x, \texttt{this} := \rho]) \uparrow a'_n$. We can compute an upper bound for a'_n by using the monotonicity of the lifting judgment and the fact that $a' \sqsubseteq (\texttt{tll}\,?\,\texttt{cmpd} : \texttt{error})$. Specifically, replacing a' with its upper bound gives us

$$\begin{aligned} &P; E \vdash S(x, (\texttt{tll}\,?\,\texttt{cmpd} : \texttt{error})[\texttt{tll} := x, \texttt{this} := \rho]) \uparrow \hat{a}'_n \\ &a'_n \sqsubseteq \hat{a}'_n \end{aligned}$$

Since $S(x, (\texttt{tll?cmpd} : \texttt{error})[\texttt{tll} := x, \texttt{this} := \rho]) = S(x, (x?\texttt{cmpd} : \texttt{error})) = \texttt{cmpd}$, we know that $a'_n \sqsubseteq \hat{a}'_n = \texttt{cmpd}$. □

E. MUTUAL EXCLUSION

We now turn our attention to mutual exclusion and show that the notion of well-formed critical sections from Appendix C.1 is preserved by reduction steps on well-typed states:

RESTATEMENT OF LEMMA 15 (MUTUAL EXCLUSION SUBJECT REDUCTION). *If* $\vdash_{cs} \Pi$ *and* $P \vdash \Pi \to \Pi'$ *then* $\vdash_{cs} \Pi'$.

PROOF. The proof is by case analysis on the evaluation rule for $P \vdash \Pi \to \Pi'$. All cases are straightforward except [RED SYNC] and [RED IN-SYNC]. In each case, let $\Pi = \langle \sigma, T \rangle$ where $n = |T|$. Since $\vdash_{cs} \Pi$, we know that $ls_i \vdash_{cs} T_i$ and $ls_i = \{\rho \mid \sigma(\rho) = \{\!|\ldots|\!\}^{i}_{c_\rho}\}$ for all $i \in 1..n$, We assume that thread k is reduced and $\Pi' = \langle \sigma', T' \rangle$ where $T'_i = T_i$ for all $i \neq k$.

—[RED SYNC]: In this case, $T_k = \mathcal{E}[\texttt{sync}\ \rho\ e]$ and $\sigma(\rho) = \{\!|\ldots|\!\}^{\perp}_{c_\rho}$. Also, $T'_k = \mathcal{E}[\texttt{in-sync}\ \rho\, e]$ and $\sigma' = \sigma[\rho \mapsto \{\!|\ldots|\!\}^{k}_{c_\rho}]$. Therefore, $ls'_k = ls_k \cup \{\rho\} = \{\rho \mid \sigma'(\rho) = \{\!|\ldots|\!\}^{k}_{c_\rho}\}$ and $ls'_k \vdash_{cs} T'_k$. Hence, we can conclude $\vdash_{cs} \Pi'$ by rule [CS STATE].

—[RED IN-SYNC]: In this case, $T_k = \mathcal{E}[\texttt{in-sync}\ \rho\ v]$ and $\sigma(\rho) = \{\!|\ldots|\!\}^{k}_{c_\rho}$. Also, $T'_k = \mathcal{E}[v]$ and $\sigma' = \sigma[\rho \mapsto \{\!|\ldots|\!\}^{\perp}_{c_\rho}]$. Therefore, $ls_k \setminus \{\rho\} \vdash_{cs} T'_k$ and $ls'_k = ls_k \setminus \{\rho\} = \{\rho \mid \sigma'(\rho) = \{\!|\ldots|\!\}^{k}_{c_\rho}\}$. Since no other ls_i changes and ρ is not held by any thread in state Π', we may conclude $\vdash_{cs} \Pi'$ by rule [CS STATE]. □

The next two lemmas show how error atomicities propagate from subexpressions to enclosing expressions.

LEMMA 29 (CONDITIONAL ERROR ATOMICITY). *If* $P; E \vdash \mathcal{E}[e] : t \cdot a$ *and* $P; E \vdash e : s \cdot (\rho\,?\,a' : \texttt{error})$ *and* $\mathcal{E} \not\equiv \mathcal{E}'[\texttt{in-sync}\ \rho\ e']$ *then* $(\rho\,?\,\texttt{const} : \texttt{error}) \sqsubseteq a$.

PROOF. Proof is by induction on $\mathcal{E}$. We show some representative cases:

—$\mathcal{E} \equiv [\ ]$: $a = \rho\,?\,a' : \texttt{error}$

—$\mathcal{E} \equiv \texttt{let}\ x\ \texttt{=}\ \mathcal{E}'\ \texttt{in}\ e'$. According to [EXP LET], the only applicable type rule,

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e] : t_1 \cdot a_1 \\ a = (a_1; a_2) \end{array}$$

for some a_2. By the inductive hypothesis, $a_1 = (\rho\,?\,a'_1 : \texttt{error})$. Thus,

$$\begin{array}{l} a = (a_1; a_2) \\ \quad \sqsupseteq (\rho\,?\,a'_1 : \texttt{error}); a_2 \\ \quad = (\rho\,?\,(a'_1; a_2) : (\texttt{error}; a_2)) \\ \quad \sqsupseteq (\rho\,?\,\texttt{const} : \texttt{error}) \end{array}$$

—$\mathcal{E} \equiv \texttt{new}_y\ c(v_{1..k-1}, \mathcal{E}', e_{k+1..n})$. According to rule [EXP NEW],

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e] : t \cdot a_k \\ a = a_1; \cdots; a_{k-1}; a_k; a_{k+1}; \cdots; a_n \end{array}$$

for some atomicities $a_{1..n}$. By the inductive hypothesis, $a_k = (\rho\,?\,a'_k : \texttt{error})$. By associativity of ';', we have

$$\begin{array}{l} \quad a \\ \sqsupseteq (a_1; \cdots; a_{k-1}); a_k; (a_{k+1}; \cdots; a_n) \\ \sqsupseteq (\rho\,?\,((a_1; \cdots; a_{k-1}); a'_k; (a_{k+1}; \cdots; a_n)) : ((a_1; \cdots; a_{k-1}); \texttt{error}; (a_{k+1}; \cdots; a_n))) \\ \sqsupseteq (\rho\,?\,\texttt{const} : \texttt{error}). \end{array}$$

—$\mathcal{E} \equiv \texttt{sync}\ \mathcal{E}'\ e'$: According to rule [EXP SYNC], $P; E \vdash_{\text{lock}} \mathcal{E}'[e]$ which requires that $P; E \vdash \mathcal{E}'[e] : s \cdot \texttt{const}$. By the inductive hypothesis, $P; E \vdash \mathcal{E}'[e] : s \cdot a'$ where $(\rho\,?\,\texttt{const} : \texttt{error}) \sqsubseteq a'$. But this is a contradiction, since $a' \neq \texttt{const}$, and this case cannot occur.

—$\mathcal{E} \equiv$ `in-sync` $\rho'\ \mathcal{E}'$ where $\rho' \neq \rho$: According to rule [EXP IN-SYNC] and the inductive hypothesis,

$$P;E \vdash \mathcal{E}'[e'] : s \cdot a'$$
$$a = IS(\rho', a')$$

By the inductive hypothesis, $(\rho\,?\,\texttt{const} : \texttt{error}) \sqsubseteq a'$, and

$$\begin{aligned} a &= IS(\rho', a') \\ &\sqsupseteq IS(\rho', (\rho\,?\,\texttt{const} : \texttt{error})) \\ &\sqsupseteq \rho\,?\,IS(\rho', \texttt{const}) : IS(\rho', \texttt{error}) \\ &\sqsupseteq (\rho\,?\,\texttt{const} : \texttt{error}) \quad \square \end{aligned}$$

LEMMA 30 (ERROR ATOMICITY). *If $P;E \vdash \mathcal{E}[e] : t \cdot a$ and $P;E \vdash e : s \cdot \texttt{error}$ then* $\texttt{error} \sqsubseteq a$.

PROOF. Proof is by induction on $\mathcal{E}$, as above. $\square$

We characterize in the following Lemmas 31 and 32 when a thread may read or write to an object's field.

LEMMA 31 (ACCESS READ). *Suppose $P \vdash \langle \sigma, T \rangle$ and $\vdash_{\text{cs}} \langle \sigma, T \rangle$ and $P \vdash_{\text{inst}}$ `class` c { $\ldots t\ fd\ g \ldots$ } and $P;E \vdash \rho : c$ and $P;E \vdash \sigma$. If T_k reads $\rho.fd$ then either:*

(1) *g is* `final`*,* `no_guard`*, or* `write_guarded_by` *l; or*
(2) *g is* `guarded_by` *l, and T_k is in a critical section on* $l[\texttt{this} := \rho]$.

PROOF. If case (1) is true, there is nothing else to prove. If case (2) is true, then we must show that T_k is in a critical section on $l[\texttt{this} := \rho]$. Since $P \vdash \langle \sigma, T \rangle$, it must be that $P;E \vdash T_k : t \cdot a$ where $a \sqsubseteq \texttt{cmpd}$ and $T_k = \mathcal{E}[\rho.fd]$. Note that $P;E \vdash \rho.fd : s \cdot (l[\texttt{this} := \rho]\,?\,\texttt{mover} : \texttt{error})$ by rule [EXP REF]. If T_k were not in a critical section on $l[\texttt{this} := \rho]$, then Lemma 29 indicates that $(l[\texttt{this} := \rho]\,?\,\texttt{const} : \texttt{error}) \sqsubseteq a$ and hence $a \not\sqsubseteq \texttt{cmpd}$, which yields a contradiction. Thus, the thread must be in a critical section on that lock. $\square$

LEMMA 32 (ACCESS WRITE). *Suppose $P \vdash \langle \sigma, T \rangle$ and $\vdash_{\text{cs}} \langle \sigma, T \rangle$ and $P \vdash_{\text{inst}}$ `class` c { $\ldots t\ fd\ g \ldots$ } and $P;E \vdash \rho : c$ and $P;E \vdash \sigma$. If T_k writes $\rho.fd$ then:*

(1) *g is* `no_guard`*; or*
(2) *g is* `guarded_by` *l or* `write_guarded_by` *l, and T_k is in a critical section on* $l[\texttt{this} := \rho]$.

PROOF. Since $P \vdash \langle \sigma, T \rangle$, it must be that $P;E \vdash T_k : t \cdot a$ where $a \sqsubseteq \texttt{cmpd}$ and $T_k = \mathcal{E}[\rho.fd = v]$. We proceed as in the previous proof, noting that g cannot be `final`, or else Lemma 30 would yield that $\texttt{error} \sqsubseteq a$. $\square$

The following lemma describes when a field access is guaranteed to be conflict-free; such conflict-free accesses commute with steps from other threads.

RESTATEMENT OF LEMMA 16 (CONFLICTING ACCESSES) *Suppose $P \vdash \langle \sigma, T \rangle$ and $\vdash_{\text{cs}} \langle \sigma, T \rangle$ and $p \vdash_{\text{inst}}$ `class` $c\{\ldots t\ fd\ g \ldots\}$ and $P;E \vdash \rho : c$ and $P;E \vdash \sigma$. If T_k accesses $\rho.fd$ then:*

(1) *g is* `final` *and* $\forall i \neq k$, *T_i does not write $\rho.fd$; or*
(2) *g is* `guarded_by` *l and* $\forall i \neq k$, *T_i does not access $\rho.fd$; or*
(3) *g is* `write_guarded_by` *l and if T_k is in a critical section on l* [`this` := ρ], *then* $\forall i \neq k$, *T_i does not write $\rho.fd$; or*
(4) *g is* `no_guard`

PROOF. Since $P \vdash \langle \sigma, T \rangle$, it must be that $P; E \vdash T_i : t_i \cdot a_i$ where $a_i \sqsubseteq \texttt{cmpd}$ for all $i \in 1..|T|$. We handle each case of g separately:

(1) g is `final`: Lemma 32 ensures that no threads write to final fields.
(2) g is `guarded_by` l: We show that no other thread can be accessing the same field. Lemmas 31 and 32 indicate that T_k must be in a critical section on $l[\texttt{this} := \rho]$. Since $\vdash_{\text{cs}} \langle \sigma, T \rangle$, we know that $\sigma(l[\texttt{this} := \rho]) = \{\!|\ldots|\!\}^k_{c'}$ for some c'. Further suppose that T_i accesses the same field, where $i \neq k$. Lemmas 31 and 32 again indicate that T_i is in a critical section on $l[\texttt{this} := \rho]$ and it would follow that $\sigma(l[\texttt{this} := \rho]) = \{\!|\ldots|\!\}^i_{c'}$. However, $i \neq k$, so this cannot occur.
(3) g is `write_guarded_by` l: The proof is similar to the previous case.
(4) g is `no_guard`: There is nothing to show in this case. □

F. CORRECTNESS OF TYPE INFERENCE

We now connect type inference to type checking by showing that if type inference succeeds, it yields an explicitly typed well-typed program.

RESTATEMENT OF THEOREM 4 (TYPE INFERENCE YIELDS WELL-TYPED PROGRAMA). *If $P \vdash \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$ then $A(P) \vdash$* wf.

PROOF. We prove this theorem by simultaneous induction on:

(1) If $P \vdash \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P) \vdash \text{wf}$.
(2) If $P \vdash E \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P) \vdash E$.
(3) If $P; E \vdash \mathit{defn} \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P); E \vdash A(\mathit{defn})$.
(4) If $P; E \vdash \mathit{meth} \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P); E \vdash A(\mathit{meth})$.
(5) If $P; E \vdash \mathit{field} \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P); E \vdash A(\mathit{field})$.
(6) If $P; E \vdash a \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P); E \vdash a$.
(7) If $P; E \vdash t \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P); E \vdash t$.
(8) If $P; E \vdash_{\text{lock}} l \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P); E \vdash_{\text{lock}} l$.
(9) If $P; E \vdash e : t \cdot d \cdot \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$, then $A(P); E \vdash e : t \cdot [\![A(d)]\!]$.

We show several interesting cases:

—[INF EXP SYNC]: If $P; E \vdash \texttt{sync}\ l\ e : t \cdot d \cdot \bar{C}$, then

$P; E \vdash_{\text{lock}} l \cdot \bar{C}_1$
$P; E \vdash e : t \cdot d' \cdot \bar{C}_2$
$d = \mathcal{S}(l, d')$

For $i \in 1..2$, $\bar{C}_i \subseteq \bar{C}$ and thus $A \models \bar{C}_i$ and A_i is well-formed for $\bar{C}_i$. The inductive hypothesis thus indicates that

$A(P); E \vdash_{\text{lock}} l$
$A(P); E \vdash e : t \cdot [\![A(d')]\!]$

These allow us to conclude, by rule [EXP SYNC], that $A(P); E \vdash e : t \cdot S(l, [\![A(d')]\!])$. Also, $S(l, [\![A(d')]\!]) = [\![A(\mathcal{S}(l, d'))]\!] = [\![A(d)]\!]$, and we are done.

—[INF METHOD]: Suppose $P; E \vdash meth \cdot \bar{C}$, where

$meth = s\ t\ md\langle \texttt{ghost}\ x_{1..n}\rangle(arg_{1..r})\ \{\ e\ \}$
$E' = E, \texttt{ghost}\ x_{1..n}, arg_{1..r}$
$P; E' \vdash e : t \cdot d \cdot \bar{C}'$
$P; E' \vdash s \cdot \bar{C}''$
$\bar{C} = (\bar{C}' \cup \bar{C}'' \cup \{lift(P, E', d) \sqsubseteq s\})$

By the inductive hypothesis,

$A(P); E' \vdash e : t \cdot [\![A(d)]\!]$

Also,

$A(meth) = A(s)\ t\ md\langle \texttt{ghost}\ x_{1..n}\rangle(arg_{1..r})\ \{\ e\ \}$
$A \models lift(P, E', d) \sqsubseteq s$

The second line above implies that $[\![lift(A(P), E', A(d))]\!] \sqsubseteq A(s)$. Note that $[\![lift(A(P), E', A(d))]\!] = a$ such that $A(P); E' \vdash [\![A(d)]\!] \uparrow a$ for some a. Thus, $a \sqsubseteq A(s)$. By Lemma 19 (7), $[\![A(d)]\!] \sqsubseteq a$, and $[\![A(d)]\!] \sqsubseteq A(s)$.

Finally, we show that $A(P); E' \vdash A(s)$. There are two cases:

—$s = a$: Since $P; E' \vdash a \cdot \bar{C}''$, the inductive hypothesis allows us to conclude $A(P); E' \vdash A(a)$.
—$s = \alpha$: Since A is a well-formed solution and $(lift(P, E', d) \sqsubseteq \alpha) \in \bar{C}$, it must be that $A(P); E' \vdash A(\alpha)$.

We may then use rule [METHOD] to conclude that $A(P); E \vdash A(meth)$. $\square$