Online Appendix to:

Specification and Static Enforcement of Scheduler-Independent Noninterference in a Middleweight Java

Zeinab Iranmanesh, Mehran S. Fallah

Contents

A The Operational Semantics of Middleweight Java
B The Type System of Multithreaded Middleweight Java
   B.1 Rules Defining Execution Time
   B.2 Typing Recursive Methods and Constructors
C Proofs

It should be noted that some definitions and references in the main paper are referred to in this document.
Appendix A. The Operational Semantics of Middleweight Java

The transition rules defining the computation of MJ expressions and statements are given in Figs. A.1-A.2. The main set of rules are shown in Fig. A.1. The function $eval(MS, x)$ returns the value of variable $x$ in the recent method scope $MS$. Moreover, the function $update(MS, x \mapsto v)$ updates the value of $x$ to $v$ in the method scope $MS$. The order of evaluation is defined by the rules in Fig. A.2. As seen, the head of FS may be an open frame containing a hole “•” which will be replaced with the value obtained from evaluating CF.

Appendix B. The Type System of Multithreaded Middleweight Java

Typing rules for expressions are in Fig. B.3. The judgment $\vdash \Delta \text{ok}$ states that the native types in the types obtained from $\Delta$ are valid in the sense that they are in compliance with $\Delta$ itself. For example, the fields of a class $C$ are of valid types according to the rule

$$
\begin{align*}
\text{dom}(\Delta_f(C)) &= \{f_1, \ldots, f_n\} \\
\Delta_f(C)(f_1) &= (C_1, t_1) \ldots \Delta_f(C)(f_n) = (C_n, t_n) \\
\Delta \vdash C_1 \text{ ok } \ldots \Delta \vdash C_n \text{ ok} \\
\Delta \vdash \Delta_f(C) \text{ ok}
\end{align*}
$$

where the premise $\Delta \vdash C_i \text{ ok}$ is defined by

$$
\frac{C \in \text{dom}(\Delta)}{\Delta \vdash C \text{ ok}}
$$

Typing rules for MMJ statements are given in Fig. B.4. The rule TS-Return is for typing $\text{return } e$:. This statement is the last statement in the body of a method $m$ whose return type is not void. Therefore, TS-Return is applied during type checking the body of $m$. At the beginning of this process, a special variable result is added to the typing context with the security level declared in the return type of $m$. As seen, this level should also be derived as the security level of $e$. Otherwise, an illegal information flow occurs from the high expression $e$ to the low value returned by invoking $m$.

The rule TS-Intro type checks a sequence of statements whose first statement is $(C, \ell) \ x$:. By appending $x : (C, \ell, H)$ to $\Gamma$, the rest of the sequence is type checked. The security level $\ell$ is also involved in the effect type $\ell_f$ because the low part of memory expands by declaring a low variable.

The typing rules for super call are shown in Fig. B.5. This construct appears in the constructor of any class. The rule T-CObject is applied when Object is the direct superclass of the class whose constructor is being type checked. For an arbitrary superclass, the rule T-CSuper, which is similar to TE-New, is applied. This is because the execution of a super call in the constructor of a class results in the invocation of the constructor of its superclass.

Finally, the rules in Fig. B.6 are to type check the whole program. For example, a constructor $C$ is well-typed if the judgment $\mathcal{L}; \Delta \vdash C \text{ cok} | \mathcal{L'}; \Delta'$ is derivable through the rule T-CBODY.
\[ \text{Figure A.1: MJ reduction rules [24].} \]
Figure A.2: MJ decomposition rules [24].
Figure B.3: Typing rules for expressions.
if \( \ell = H \times L \times L \times L \), \( \Delta; \Gamma; \) then
\[
\begin{array}{l}
\Delta; \Gamma; \vdash e_1 : (C, \ell, \ell') | T_1
\Delta; \Gamma; \vdash e_2 : (C, \ell, \ell') | T_2
\Delta; \Gamma; \vdash s_1 : (void, \ell, \ell') | T'_1
\Delta; \Gamma; \vdash s_2 : (void, \ell, \ell') | T'_1
\end{array}
\]
\[
\begin{array}{l}
\ell_e \leq \ell'_e \cap \ell^o_e \quad \ell'_f = \ell^o_e \cap \ell^o'_e
\end{array}
\]
if \( \ell_e = H \) then\( \Delta; \Gamma; \Rightarrow \Delta; \Gamma; \)
\[
\begin{array}{l}
\Delta; \Gamma; \vdash nil \mid time(\ell_1) = t \wedge \Delta; \Gamma; \vdash nil \mid time(\ell_2) = t' \wedge t \neq t'
\Delta; \Gamma; \vdash nil \mid time(\ell_1) = t_{\text{nil}} \wedge \Delta; \Gamma; \vdash nil \mid time(\ell_2) = t_{\text{nil}}
\end{array}
\]
then \( T' = T'_1 \cup \{ t_H \} \) else \( T' = T'_2 \)
\[
\begin{array}{l}
\Delta; \Gamma; \vdash if (e_1 == e_2) \{ s_1 \} \text{ else } \{ s_2 \} : (void, \ell, \ell') | T'
\end{array}
\]
\[
\begin{array}{l}
\ell; \Delta; \Gamma; \vdash e' : (C, \ell, \ell') | T_1
\Delta; \Gamma; \vdash e : (C, \ell, \ell') | T_2
\end{array}
\]
\[
\begin{array}{l}
\Delta_f(C_1)(f) = (C_2, \ell_2) \quad \ell_1 \leq \ell_2 \quad \ell'_2 = \ell^o_2 \cap \ell_2
\end{array}
\]
if \( \ell_2 = L \) then \( T' = T_2 \cup \{ w \} \) else \( T' = T_2 \)
\[
\begin{array}{l}
\ell; \Delta; \Gamma; \vdash e'.f = e : (void, \ell, \ell') | T'
\end{array}
\]
\[
\begin{array}{l}
\ell; \Delta; \Gamma; \vdash \text{fieldWrite} x \neq \text{this} \quad \ell; \Delta; \Gamma; \vdash e : (C, \ell, \ell') | T
\end{array}
\]
\[
\begin{array}{l}
\Gamma = \epsilon, \text{result} : (C, \ell, H), \circ \quad \ell; \Delta; \Gamma; \vdash e : (C, \ell, \ell') | T'
\end{array}
\]
\[
\begin{array}{l}
\ell; \Delta; \Gamma; \vdash return e : (C, H, \ell') | T'
\end{array}
\]
\[
\begin{array}{l}
\ell; \Delta; \Gamma; \vdash s_1 \ldots s_n : (void, \ell, \ell') | T'
\ell; \Delta; \Gamma; \vdash \{ s_1 \ldots s_n \} : (void, \ell, \ell') | T'
\ell; \Delta; \Gamma; \vdash (C, \ell) \times s_1 \ldots s_n : (\tau, \ell'_f, \ell') | T'
\end{array}
\]
\[
\begin{array}{l}
\ell; \Delta; \Gamma; \vdash s_1 \ldots s_n : (\tau, \ell', \ell') | T'
\ell; \Delta; \Gamma; \vdash s_1.s_2 \ldots s_n : (\tau, \ell', \ell') | T'
\ell; \Delta; \Gamma; \vdash s : \eta | T'
\ell; \Delta; \Gamma; \vdash s : \eta' | T''
\ell; \Delta; \Gamma; \vdash if (\eta \neq \eta') \{ T' \} \subset T''
\end{array}
\]
Figure B.4: Typing rules for statements.
The body of method \( m \) in class \( C \) is type checked by the rule T-MDefn. As seen, the context \( \Gamma_1 \) consists of the parameters to \( m \) and a special variable \textbf{result} representing the return value of \( m \) all mapped to their types as obtained from the initial \( \Delta \). If type checking is successful with considering high security level for variable \textbf{this}, the level \( \ell_m \) in the type of \( m \) is set to \( H \). This level is set to \( L \) if \( m \) is typable with respect to variable \textbf{this} of low security level. If the method being checked represents the main thread, any access to a low field in the body of the method indeed occurs in a thread. In this case, \( \ell^w \) and \( \ell^r \) are inscribed in the set \( \mathcal{T}_m \) if a low field is modified or read in the body of \( m \). Similarly, \( \ell^H \) is inscribed in \( \mathcal{T}_m \) if the execution time of \( m \) may depend on high values. Moreover, the symbol \( \triangleright \) may also be added to \( \mathcal{T}_m \) through the function \textit{conflict}. The rule T-CDefn is similar to T-MDefn.

\textbf{Appendix B.1. Rules Defining Execution Time}

The execution time of MMJ components are defined by the rules given in Figs. B.7-B.9. The sequence \( (C, m) \) of pairs of class-method and class-constructor names is for detecting loops. Moreover, the judgment \( \Delta : \Gamma \vdash^t e : C \) is used to show that the native type of \( e \) is \( C \). This judgment is the same as the typing judgment of MMJ except that the components other than those concerning native types are removed. In Fig. B.7, the function methodTime is used in the rule given for the invocation of a method \( m \) on an expression of type \( C \). This function returns \( t_{nil} \) if no unique execution time can be derived for \( m \) in the subclasses of \( C \).

\[
\text{methodTime} \quad (\Delta, C, (C, m), m, C) = \begin{cases} 
  t & \text{if } \exists t. \forall C_1, C_2. \\
  (C_1 < C \land C_2 < C) \Rightarrow \exists x, \Gamma_1, \Gamma_2, mbody(C, m) = (x, \emptyset) \land \Gamma_1 = x : C \land \\
  \Gamma_1 \vdash \textsf{this} : C_1 \land \Gamma_2 \vdash \textsf{this} : C_2 \land \\
  \Delta : \Gamma_1 \vdash (C, m), (C_1, m) \vdash \text{time}(C_1)(m) = t \land \\
  \Delta : \Gamma_2 \vdash (C, m), (C_2, m) \vdash \text{time}(C_2)(m) = t \\
  t_{nil} & \text{otherwise}
\end{cases}
\]
\[ \Delta_m(C)(m) = \circ, \kappa_1, \ldots, \kappa_n \rightarrow \kappa \]

\[ \text{mbody}(C, m) = (x_1, \ldots, x_n, \bar{s}) \]

\[ \Gamma_1 = x_1 : \kappa_1, \ldots, x_n : \kappa_n, \text{result} : (\kappa, \tau, \kappa, \ell, H) \]

\[ \ell; \Delta, \Gamma_1, \text{this} : (C, H, H); \emptyset \vdash \bar{s} : (\kappa, \tau, \ell, \ell_1') | \ell_1' \land \ell_1 = L \]

\begin{align*}
\text{if} & \quad (m \neq \text{run}) \Rightarrow (\forall C' < C. \left( (C', m) \in \ell \land \Delta_m(C')(m) = \ell'_0 \rightarrow \circ \Rightarrow \ell'_0 = H \right) ) \\
\text{then} & \quad \ell_1 = H \quad \text{else} & \quad (L; \Delta, \Gamma_1, \text{this} : (C, L, H); \emptyset \vdash \bar{s} : (\kappa, \tau, \ell, \ell' | \ell_1' \land \ell_1 = L) \\
\text{if} & \quad \ell_1 = H \quad \text{then} & \quad (\ell'_0 = \ell_1' \land T = T_1) \quad \text{else} & \quad (\ell'_0 = \ell_1' \land T = T_2) \\
\text{if} & \quad m = \text{main} \quad \text{then} & \quad T' = \{s | H \in T \} \cup \text{conflict}(T, \{w | w \in T \} \cup \{v | v \in \ell \}) \quad \text{else} & \quad T' = T \\
\Delta' = \Delta \cup \Delta_m(C)(m) \rightarrow \ell_1, \kappa_1, \ldots, \kappa_n \rightarrow (\ell_0', \ell_0' | \ell') \rightarrow \kappa \]
\end{align*}

\[ \ell; \Delta \vdash \text{chBody}(C, m) \text{ ok} | \Delta' \quad \text{(T-MDEFN)} \]

\begin{align*}
m' = \text{method}(m) & \quad \text{if} & \quad (C, m') \notin \ell \quad \text{then} & \quad (L; \Delta \vdash \text{chBody}(C, m') \text{ ok} | \Delta' \land \ell' = \ell \cup \{(C, m')\}) \\
& \quad \text{else} & \quad (\ell' = \ell \land \Delta' = \Delta) \\
\ell; \Delta \vdash \text{mbody}(C, m) \text{ ok} | \ell'; \Delta' \quad \text{(T-MBODY)} \]
\end{align*}

\begin{align*}
\Delta_1(C) = \circ, \kappa_1, \ldots, \kappa_n, (\circ) \\
\text{cnbody}(C) = (x_1, \ldots, x_n, \text{super}(\varepsilon); \bar{s}) \\
\Gamma_1 = x_1 : \kappa_1, \ldots, x_n : \kappa_n \\
\text{if} & \quad (L; \Delta, \Gamma_1, \text{this} : (C, H, H); \emptyset \vdash \text{super}(\varepsilon); : (\text{void}, \ell_1', \ell_1') | \ell_1' \land \ell_1 = H) \\
& \quad \text{then} & \quad \ell_1 = H \quad \text{else} & \quad (L; \Delta, \Gamma_1, \text{this} : (C, L, H); \emptyset \vdash \bar{s} : (\text{void}, \ell', \ell') | \ell_1' \land \ell_1 = L) \\
& \quad \text{if} & \quad \ell_1 = H \quad \text{then} & \quad (\ell'_0 = \ell_1' \land \ell_1 = L) \quad \text{else} & \quad (\ell'_0 = \ell_1' \land \ell_1 = L) \\
& \quad \text{if} & \quad \bar{s} = \text{nil} \quad \text{then} & \quad (\ell'_0 = H \land T = \{a \in T_1 | \ell_1 = H \} \cup \{a \in T_2 | \ell_1 = L \}) \\
& \quad \text{else} & \quad (\Delta'_0 = \Delta_1(C) \rightarrow \ell_1, \kappa_1, \ldots, \kappa_n, (\ell'_0 | \ell') \rightarrow \kappa) \\
\ell; \Delta \vdash \text{consBody}(C) \text{ ok} | \Delta' \quad \text{(T-CDefN)} \]
\end{align*}

\begin{align*}
\text{if} & \quad C \notin \ell \quad \text{then} & \quad (L; \Delta \vdash \text{consBody}(C) \text{ ok} | \Delta' \land \ell' = \ell \cup \{C\}) \\
& \quad \text{else} & \quad (\ell' = \ell \land \Delta' = \Delta) \\
\ell; \Delta \vdash C \text{ ok} | \ell'; \Delta' \quad \text{(T-CBody)} \\
& \quad \text{dom}(\Delta) = \{C_1, \ldots, C_n\} \\
& \quad \text{dom}(\Delta_m(C_i)) = \{m^C_i \text{ to } m^{C_i}\} \quad (i = 1, \ldots, n) \\
& \quad g = n + k_{C_1} + \ldots + k_{C_n} \\
& \quad \ell_1 = \{\text{Object} \} \\
& \quad \Delta_1 = \Delta_m \cup \{\text{Object}, (H, \text{nil}, (H, \emptyset)), \Delta_f\} \\
& \quad S = \bigcup_{j=1}^{r} \{\text{mbody}(C_j, m_j^{C_i}) \text{ ok} | C_j \text{ ok} | \{j = 1, \ldots, k_{C_i}\} \} \\
& \quad \exists s_1, \ldots, s_g \in S. \quad \ell_1; \Delta_1 \vdash \ell'_1 | \ell; \Delta_2 | \ell_2; \ldots; \ell_{g} | \Delta_2 \vdash \Delta_{g+1} (\text{if} \quad i \neq j \text{ then} \quad s_i \neq s_j) \\
& \quad \ell'_1 = \Delta_{g+1} \land \forall 1 \leq i \leq n, 1 \leq j \leq k_{C_i}, \text{chTP}(\Delta', m_j^{C_i}, C_i) \\
& \quad T = \{a \in T'_m | C \in \text{dom}(\Delta') \land m \neq \text{start} \land \Delta_m(C)(m) = \circ \rightarrow (\circ, T_m^{C_i}) \rightarrow \circ\} \\
& \quad \ell' = T' \cup \{a \in T_1 | C \in \text{dom}(\Delta') \land \Delta_m(C) = \circ \rightarrow (\circ, T_1) \rightarrow \circ\} \\
& \quad \Delta \vdash P \text{ ok} \quad \text{(T-ProgDef)} \]

Figure B.6: Typing rules for programs.
\[ \Delta; \Gamma; (C, m) \vdash time(x) = 1 \]

\[ \Delta; \Gamma; (C, m) \vdash time(\textbf{null}) = 0 \]

\[ \Delta; \Gamma; (C, m) \vdash time(e) = t \]

\[ \Delta; \Gamma; (C, m) \vdash time(e.f) = t + 3 \]

\[ \Delta; \Gamma; (C, m) \vdash time(e) = t \]

\[ \Delta; \Gamma; (C, m) \vdash time(C.e) = t + 3 \]

\[ \Delta; \Gamma; (C_1, m_1), \ldots, (C_k, m_k) \vdash time(e) = t \]

\[ \Delta; \Gamma; \vdash e : C \]

\[ \Delta; \Gamma; (C_1, m_1), \ldots, (C_k, m_k) \vdash time(e_1) = t_1 \]

\[ \ldots \]

\[ \Delta; \Gamma; (C_1, m_1), \ldots, (C_k, m_k) \vdash time(e_n) = t_n \]

\[ \text{if } \exists 1 \leq i \leq k. (C_i < C \land m_i = m) \text{ then } t' = t_{\text{null}} \text{ else} \]

\[ \text{if } m = \text{start} \text{ then } t' = 0 \text{ else} \]

\[ \Delta; \Gamma; \vdash e_1 : C'_1 \]

\[ \ldots \]

\[ \Delta; \Gamma; \vdash e_n : C'_n \]

\[ t' = \text{methodTime}(\Delta, C; (C_1, m_1), \ldots, (C_k, m_k), m, C'_1, \ldots, C'_n) \]

\[ \Delta; \Gamma; (C_1, m_1), \ldots, (C_k, m_k) \vdash time(e.m(e_1, \ldots, e_n)) = t + \sum_{j=1}^{n} t_j + t' + 2n + 3 \]

\[ \Delta; \Gamma; (C_1, m_1), \ldots, (C_k, m_k) \vdash time(e) = t_1 \]

\[ \ldots \]

\[ \Delta; \Gamma; (C_1, m_1), \ldots, (C_k, m_k) \vdash time(e_n) = t_n \]

\[ \text{if } \exists 1 \leq i \leq k. m_i = C \text{ then } t' = t_{\text{null}} \text{ else} \]

\[ \Delta; \Gamma; \vdash e_1 : C'_1 \]

\[ \ldots \]

\[ \Delta; \Gamma; \vdash e_n : C'_n \]

\[ \text{cnbody}(C) = (x_1, \ldots, x_n, s) \]

\[ \Gamma'' = x_1 : C'_1, \ldots, x_n : C'_n, \text{this : C} \]

\[ \Delta; \Gamma''; (C_1, m_1), \ldots, (C_k, m_k), (C, C) \vdash time(C) = t' \]

\[ \Delta; \Gamma; (C_1, m_1), \ldots, (C_k, m_k) \vdash time(\text{new} (C, l)(e_1, \ldots, e_n)) = \sum_{j=1}^{n} t_j + t' + 2n + 3 \]

Figure B.7: Execution time for MMJ expressions.
\[ \Delta; \Gamma; (C, m) \vdash \text{time}() = 1 \]

\[ \Delta; \Gamma; (C, m) \vdash \text{time}(e) = t \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(e) = t + 2 \]

\[ \Delta; \Gamma; (C, m) \vdash \text{time}(e) = t \quad \Delta; \Gamma; (C, m) \vdash \text{time}(e') = t' \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(e.f = e') = t + t' + 0 \]

\[ \Delta; \Gamma; (C, m) \vdash \text{time}(x = e) = t + 4 \]

\[ \Delta; \Gamma; (C, m) \vdash \text{time}(\text{return } e) = t + 4 \]

\[ \Delta; \Gamma; (C, m) \vdash \text{time}(s_1 \ldots s_n) = t \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(\{s_1 \ldots s_n\}) = t + 4 \]

\[ \Delta; \Gamma, x : (C, \ell, H); (C, m) \vdash \text{time}(s_1 \ldots s_n) = t \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(\{s_1 \ldots s_n\}) = t + 2 \]

\[ s_1 \neq (C, \ell) x; \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(s_1) = t_1 \quad \Delta; \Gamma; (C, m) \vdash \text{time}(s_2 \ldots s_n) = t \]
\[ \text{if } s_1 = e.\text{start}(); \text{ then } t' = t - 1 \text{ else } t' = t \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(s_1 \ldots s_n) = t_1 + t' \]

\[ \Delta; \Gamma; (C, m) \vdash \text{time}(e_1) = t_1 \quad \Delta; \Gamma; (C, m) \vdash \text{time}(e_2) = t_2 \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(\{s_1\}) = t_1 \quad \Delta; \Gamma; (C, m) \vdash \text{time}(\{s_2\}) = t_2 \]
\[ \text{if } t_1^* = t_2^* \text{ then } t^* = t_1^* \text{ else } t^* = t^* \]
\[ \Delta; \Gamma; (C, m) \vdash \text{time}(\text{if } e_1 = e_2 \{s_1\} \text{ else } \{s_2\}) = t_1^* + t_2^* + t^* + 5 \]

Figure B.8: Execution time for MMJ statements.
For an invocation, a loop is detected if the method or constructor invoked is a member of the loops collected in the set $\mathcal{R}$. This set is obtained by investigating the body of $m$ through LOOPDET. As seen, the hypotheses of the judgment defined by LOOPDET include $\Gamma$ which is initially considered to be a mapping from parameters to $m$ to their types.

The sequence $(C, m_1), (C, m_2), \ldots, (C, m_n)$ in LOOPDET is $(C, m')$ when this rule is initially applied through METHODLOOP, and in turn, through T-MBODY-LOOP. A pair $(C_{n+1}, m_{n+1})$ is appended to this sequence if method $m_{n+1}$ of class $C_{n+1}$ is invoked in the body of the last element of the sequence. The body of the new method or constructor should be investigated through the same rule. In LOOPDET, all invocations of methods and constructors that appear in $\bar{s}$ are taken into account. For an invocation, a loop is detected if the method or constructor invoked is $m_1$ of $C_1$. If so, the loop is added to the set $\mathcal{R}$ corresponding to that invocation.

Figure B.9: Execution time for MMJ super call, methods, and constructors.

Appendix B.2. Typing Recursive Methods and Constructors

The rules T-MBODY-LOOP and T-CBODY-LOOP in Fig. B.10 are devised to derive the types of methods and constructors that make a loop of invocations. These rules are based on some judgments defined in Fig. B.11. The rule T-MBODY-LOOP derives the type of method $m$ of class $C$ if it is not known yet, i.e., $(C, m') \notin \mathcal{L}$ where $m'$ is method$(m)$. It first checks that $m'$ is a member of some loop of invocations. In fact, the judgment $\Delta; (C_1, m_1), \ldots, (C_n, m_n) \vdash loop(C, m)/\mathcal{R}; \mathcal{R}'$ defined by METHODLOOP in Fig. B.11 states that method $m$ of class $C$ is a member of the loops collected in the set $\mathcal{R}$. This set is obtained by investigating the body of $m$ through LOOPDET. As seen, the hypotheses of the judgment defined by LOOPDET include $\Gamma$ which is initially considered to be a mapping from parameters to $m$ to their types.

The sequence $(C_1, m_1), (C_2, m_2), \ldots, (C_n, m_n)$ in LOOPDET is $(C, m')$ when this rule is initially applied through METHODLOOP, and in turn, through T-MBODY-LOOP. A pair $(C_{n+1}, m_{n+1})$ is appended to this sequence if method $m_{n+1}$ of class $C_{n+1}$ is invoked in the body of the last element of the sequence. The body of the new method or constructor should be investigated through the same rule. In LOOPDET, all invocations of methods and constructors that appear in $\bar{s}$ are taken into account. For an invocation, a loop is detected if the method or constructor invoked is $m_1$ of $C_1$. If so, the loop is added to the set $\mathcal{R}$ corresponding to that invocation.
\[
\forall s_1, s_2 \in R.
\]

\[
\begin{align*}
&\begin{cases}
\text{if } (C, m') \notin \mathcal{L} \text{ then} \\
\Delta_1(C, m') \vdash \text{loop}(C, m')|\mathcal{R}; \mathcal{R}' \land \mathcal{R} \neq \emptyset \\
\forall 1 \leq i \leq k_1, 1 \leq j \leq k_2. \quad (C_i \equiv C' \land m_i = m'_j) \Rightarrow (C_i = C \land m_i = m') \\
\Delta_m(C)(m') = \top, \kappa_1, \ldots, \kappa_n \Rightarrow \kappa \\
\Delta_1 = \Delta[\Delta_m(C)(m') \rightarrow \mathcal{H}, \kappa_1, \ldots, \kappa_n \rightarrow \mathcal{H}, \emptyset] \\
\Delta' = \text{equalTP}(\mathcal{R}, \mathcal{L}, \Delta_1) \\
\{C_i | (C_i, m_1), \ldots, (C_i, m_k) \in \mathcal{R} \} \cup \\
\{((C_i, m_1),\ldots,(C_i, m_k)) \in \mathcal{R} \land m_i \neq C_i \}\end{cases}
\end{align*}
\]

\[
\mathcal{L}' = \mathcal{L} \cup \left( (\mathcal{L}', \Delta') \text{ ok}[\mathcal{L}', \Delta'] \right)
\]

(T-MBODY-LOOP)

\[
\forall s_1, s_2 \in R.
\]

\[
\begin{align*}
&\begin{cases}
\text{if } C \notin \mathcal{L} \text{ then} \\
\Delta_1(C, C) \vdash \text{loop}(C, C)|\mathcal{R}; \mathcal{R}' \land \mathcal{R} \neq \emptyset \\
\forall 1 \leq i \leq k_1, 1 \leq j \leq k_2. \quad (C_i \equiv C' \land m_i = m') \Rightarrow (C_i = C \land m_i = C') \\
\Delta_m(C) = \top, \kappa_1, \ldots, \kappa_n \Rightarrow \kappa \\
\Delta_1 = \Delta[\Delta_m(C) \rightarrow \mathcal{H}, \kappa_1, \ldots, \kappa_n \rightarrow \mathcal{H}, \emptyset] \\
\Delta' = \text{equalTP}(\mathcal{R}, \mathcal{L}, \Delta_1) \\
\{C_i | (C_i, m_1), \ldots, (C_i, m_k) \in \mathcal{R} \} \cup \\
\{((C_i, m_1),\ldots,(C_i, m_k)) \in \mathcal{R} \land m_i \neq C_i \}\end{cases}
\end{align*}
\]

\[
\mathcal{L}' = \mathcal{L} \cup \left( (\mathcal{L}', \Delta') \text{ ok}[\mathcal{L}', \Delta'] \right)
\]

(T-CBODY-LOOP)

Figure B.10: Typing rules for recursive methods and constructors.
\[ mbody(C, m) = (x_1, \ldots, x_n, \bar{s}) \]
\[ \Delta_m(C)(m) = o, \kappa_1, \ldots, \kappa_n \rightarrow \kappa \]
\[ \Gamma_1 = x_1 : k_1, \tau, \ldots, x_n : \kappa_n, \tau, \text{this} : C \]
\[ \Delta; \Gamma_1(C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}({\bar{s}})R; R' \]
\[ \Delta(C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}(C, m)R; R' \]

\[ \text{methodLoop} \]

\[ cbody(C) = (x_1, \ldots, x_n, \bar{s}) \]
\[ \Delta_c(C) = o, \kappa_1, \ldots, \kappa_n, o \]
\[ \Gamma_1 = x_1 : k_1, \tau, \ldots, x_n : \kappa_n, \tau, \text{this} : C \]
\[ \Delta; \Gamma_1(C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}({\bar{s}})R; R' \]
\[ \Delta(C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}(C, C)R; R' \]

\[ \text{consLoop} \]

\[ \langle \bar{s} \rangle = \circ (C'_1, \ell_1) \ x'_1; \ldots ; \circ (C'_k, \ell_k) \ x'_k; \circ e.m'_i(e) \bigwedge m' = \text{method}(m'_1) \bigwedge \Delta; \Gamma, x'_1 : (C'_1, \ell_1, H), \ldots, x'_k : (C'_k, \ell_k, H) \vdash e : C' \]
\[ \langle \bar{s} \rangle = \circ \text{super}(e) \bigwedge \Delta_1 = \circ \text{this} : C'_1 \land m' = C' \]
\[ \text{if } (C' = C_1 \land m' = m_1) \text{ then } \mathcal{R}_{(C', m')} = \{(C_2, m_2), \ldots, (C_n, m_n), (C_1, m_1)\} \text{ else } \mathcal{R}_{(C', m')} = \{(C', m')\} \]
\[ \text{if } \exists ! i < n. (C' = C_i \land m' = m_i) \text{ then } \mathcal{R}'_{(C', m')} = \{(C', m')\} \text{ else } \mathcal{R}'_{(C', m')} = \emptyset \]
\[ \hat{\mathcal{R}} = \bigcup_{C', m'} \{(C, m) \mathcal{R}_{(C', m')} \downarrow \land (C, m) \in \mathcal{R}_{(C', m')}\} \]
\[ \hat{\mathcal{R}}' = \bigcup_{C', m'} \{(C, m) \mathcal{R}'_{(C', m')} \downarrow \land (C, m) \in \mathcal{R}'_{(C', m')}\} \]

\[ \text{loopDet} \]

\[ \Delta; \Gamma_1(C_1, m_1), (C_2, m_2), \ldots, (C_n, m_n) \vdash \text{loop}({\bar{s}})R; R' \]

\[ \text{loopType} \]

\[ \Delta; \Delta \vdash \text{loopType} \mathcal{R} \end{align*} \]

\[ \mathcal{A} = \{(C', m')| (C', m') \in \mathcal{A}' \land m' \neq C'\} \cup \{C'| (C', C') \in \mathcal{A}'\} \]
\[ \mathcal{A}; \ell; \Delta \vdash \text{chBody}(C, m) \end{align*} \]

\[ \text{methodBody} \]

\[ \mathcal{A} = \{(C', m')| (C', m') \in \mathcal{A}' \land m' \neq C'\} \cup \{C'| (C', C') \in \mathcal{A}'\} \]
\[ \mathcal{A}; \ell; \Delta \vdash \text{consBody}(C) \end{align*} \]

\[ \text{consBody} \]

Figure B.11: Some judgments that appear in the rules for type checking recursive methods and constructors. The symbol “\(o\)” stands for wild card and “\(\bar{s}\)” for “already derived.”
As will be seen, the types of all constructors and methods in \( R \) are accessible through \( \Delta' \). Therefore, corresponding elements are added to the set \( \mathcal{L} \). The
function equalTP utilizes the judgment \( \mathcal{L}; \Delta \vdash \text{loopType}(\mathcal{R})|\Delta' \) which is defined by LOOPType in Fig. B.11. This rule derives the types of all methods and constructors in \( \mathcal{R} \) and modifies \( \Delta \) in such a way that the new types can be accessed through \( \Delta' \). Note that the last pair of any sequence in \( \mathcal{R} \) is the entry point. As an example, assume \( \mathcal{R} = \{(C_1, m_1), (C, m'), ((C_2, C_2), (C, m'))\} \). Here, \( m' \) makes one loop with \( m_1 \) and another loop with \( C_2 \).

In LOOPType, the members of any sequence in \( \mathcal{R} \), except for their last members, are type checked in reverse order. In fact, the penultimate member of any sequence is type checked first using METHODBODY or CONSBODY, and in turn, through the rule T-MDEFN or T-CDEFN in Fig. B.6. Indeed, a set \( \mathcal{A} \) consisting of those constructors and methods whose types have been partially derived should be added to the hypotheses of the judgments defined by these rules. Similarly, such a set should also be added to the hypotheses of the typing judgment of MMJ. Moreover, the typing rules TE-METHOD, TE-NEW, and T-CSUPER should be modified in such a way that the method or constructor invoked is checked to be a member of \( \mathcal{L} \) or \( \mathcal{A} \). The modified TE-METHOD is shown in Fig. B.12—the other rules can be modified similarly, but we do not repeat them here for brevity. Note that as T-MDEFN and T-CDEFN are also applied through T-MBODY and T-CBODY in Fig. B.6, \( \mathcal{A} \) should be the empty set in the corresponding judgments.

As seen in LOOPType, the type of the \( j \)th, except for the last, member of the \( i \)th sequence \( s_i \) is accessible through mapping \( \Delta^i_j \). This mapping is part of the context in type checking the previous members of the sequence. In this way, \( \Delta^i_1 \) contains the types of all methods and constructors in \( s_i \) except the last one. This mapping is then considered as the initial mapping in type checking the members of \( s_{i+1} \). Therefore, \( \Delta^i_{n_i} \) contains the types of all methods and constructors in \( \mathcal{R} \) except the entry point to the loops in this set. This last member of all sequences is finally type checked against \( \Delta^i_n \) and the set \( \mathcal{A} \) consisting of the first members of all sequences. The resulting mapping \( \Delta' \) contains the types derived for all methods and constructors in \( \mathcal{R} \). Considering the function equalTPlp again, it returns \( \Delta' \) if this mapping is the same as the initial \( \Delta \) considered in LOOPType. In truth, this occurs when the types derived for the methods and constructors in \( \mathcal{R} \) match their bodies. If this is not the case, the rule is applied again with the initial mapping \( \Delta' \).

Appendix C. Proofs

Lemma 1. If \( \mathcal{A}; \mathcal{L}; \Delta; \Gamma; T \vdash e : (\tau, \ell, \ell^o)|T' \), then

1. if \( \ell^o = H \) and \( (H_1, TP_1, S^1_{CFG}) \rightarrow (H_2, TP_2, S^2_{CFG}) \) appears in an evaluation of \( e \), then \( H_1 =_L H_2 \) and \( \text{dom}(TP_1) = \text{dom}(TP_2) \), and

2. if \( \ell = L \) and \( (H, TP, S_{CFG}) \rightarrow (H_1, TP_1, S^1_{CFG}) \) and \( (H_2, TP_2, S^2_{CFG}) \rightarrow (H', TP', S'_{CFG}) \) are the first and last transitions in an evaluation of \( e \) with \( TP(th) = (VS, e, FS) \) and \( TP'(th) = (VS', v, FS) \) for some thread identifier \( th \), then the value \( v \) is either null or \( H^*_S(v) = L \) where \( H^*_S(v) \) is the security level of the object pointed by \( v \).
Proof. 1. Note that the effect type $H$ cannot be derived for $e$ through subtyping on expression types if the effect type $L$ is derived for this expression. The proof is by induction on the structure of $e$.

- $x$: As a nonpromotable expression, its effect type is derived as $H$ and the evaluation of variable $x$ takes only one step, $run_{th}$, which does not change heap and thread pool.

- null: Any evaluation of this expression is nil.

- $e'.f$: By the rule TE-FIELDACCESS, the effect type of this expression is the effect type of $e'$. The evaluation of $e'.f$ is composed of evaluating $e'$ to an object identifier $o$ and the evaluation of $o.f$ afterward. By induction hypothesis on $e'$, the transitions in its evaluation do not change the low part of heap and domain of thread pool. The evaluation of $o.f$ takes only one step, $run_{th}$, which does not change heap and thread pool.

- $(C)e'$: Similar to $e'.f$.

- $e'.m(e_1,\ldots,e_n)$: As $e' = H$, method $m$ cannot be start according to the rule TE-METHOD. Moreover, the effect types of expressions $e'$ and $e_1,\ldots,e_n$ should also be $H$. By induction hypothesis on these expressions, the transitions in their evaluations do not change the low part of heap and domain of thread pool. According to TE-METHOD, the effect type of $m$ in the type (class) of $e'$ should also be $H$. To evaluate $o.m(v_1,\ldots,v_n)$, the body of $m$ in the type (class) of $o$ is executed. As $e''_m = H$ in the type (class) of $e'$ as well as its subclasses according to the type system of MMJ, the claim is proved by Lemma 3. It can be easily shown that the native type of $o$ is related, by subclassing, to that of $e$.

- new $(C,\ell)(e_1,\ldots,e_n)$: Main parts of the proof for this case are similar to the previous one, although Lemma 4 is utilized here. Moreover, note that $\ell = H$ by the rule TE-New. In this way, a high object is created and the low part of heap is not changed.

2. Note that the security level $L$ cannot be derived for $e$ through subtyping on expression types if the security level $H$ is derived for this expression. The proof is by induction on the structure of expressions whose values compose the value of $e$, as determined for each case.

- $x$: The value of $x$ is obtained in one step of $run_{th}$ by the function $eval(MS,x)$ which returns the value of $x$ in the recent method scope $MS$. The value $v$ has been assigned to $x$ before this step. It can be through E-Translate, and in turn, the transitions E-VarWrite, E-VarIntro, E-New, E-Super, E-Method or E-MethodVoid (see Appendix A). For the closed frame $x = v$, the rule E-VarWrite sets $v$ as the value of $x$. Note that $x = e'$; is type checked by the rule TS-VARWRITE which forces $e'$ to be low if the level of $x$ is $L$. By
induction hypothesis on $e'$, either the value $v$ is null or its security level is $L$. The transition E-VarIntro sets null as the value of $x$. The transitions E-New, E-Super, E-Method, and E-MethodVoid set the value passed as an actual parameter to formal parameter $x$—the MMJ semantic rule E-MethodVoidStart sets the value of variable this similarly. Constructor invocations, super calls, and method invocations are checked by typing rules TE-New, T-CSuper, and TE-Method, respectively. In these rules, if the security level of formal parameter $x$ is $L$ in the type of a constructor or method, the security level of the expression $e'$ passed as the corresponding actual parameter is also forced to be $L$. By induction hypothesis on $e'$, the value of $e'$ is either null or its security level is $L$. A similar justification can be given for the variable this.

- **null**: Immediate.

- **$e'.f$**: According to the rule TE-FieldAccess, the security level of this expression is $L$ if the security level of both $e'$ and $f$ is $L$. The evaluation of $e'.f$ consists of evaluating $e'$ to an object identifier $o$ and evaluating $o.f$ afterward. By induction hypothesis on $e'$, the security level of $o$ is $L$. The evaluation of $o.f$ to a value $v$ takes one step of runth. The value $v$ has been assigned to $o.f$ before this step. This can be through an MJ transition E-FieldWrite or E-New. The former is applied when the closed frame is $o.f=v$. The object identifier $o$ is the value of an expression $e''$. The statement $e''.f = e_1$; where $e_1$ has been evaluated to $v$ is type checked by the rule TS-FieldWrite. According to this rule, the security level of $e''$ cannot be $H$ because the field $f$ is low. Moreover, the security level of $e_1$ is also $L$. Therefore, its value $v$ is of security level $L$ by the induction hypothesis. The MJ transition E-New assigns null as the value of the fields of the object that is created.

- **$(C)e'$**: By induction hypothesis on $e'$, its value which is also the value of $e$ is low. Note that if the value of $e'$ is null, the value of $e$ is also null.

- **$e'.m(e_1,\ldots,e_n)$**: According to the typing rule TE-Method, the security level of this expression is $L$ if both the security level of $e'$ and the security level in the return type of $m$ are $L$. The value of this expression is either the value of $e'$, if the native type in the return type of $m$ is void, or the value of $e''$ in return $e''$; as the last statement of $m$. By induction hypothesis on $e'$, the security level of its value is $L$. According to the typing rule TS-Return, the security level of $e''$ is also $L$ because the security level of the variable result is set to the security level in the return type of $m$. By induction hypothesis on $e''$, the security level of its value is $L$.

- **new $(C,\ell)(e_1,\ldots,e_n)$**: The value of this expression is the identifier of the new object just created. As the level $\ell$ declared in the expression is $L$ according to TE-New, an object with a low identifier is created.
Lemma 2. Assume that $A; L; Δ; Γ; T ⊢ s : (τ, ℓ^o, ℓ^e)|T'$ and $(H_1, TP_1, S_{CFG}^1) → (H_2, TP_2, S_{CFG}^2)$ appears in an execution $S$ of statement $s$.

1. If $ℓ^o = H$, then $H_1 =_L H_2$ and $\text{dom}(TP_1) = \text{dom}(TP_2)$.

2. If $ℓ^e = H$ and $(H, TP, S_{CFG}) → (H', TP', S_{CFG}')$ is the first transition in $S$ such that $TP(th) = (VS_{th}, S, FS)$ for some thread identifier $th$, then $TP_1(th) = (VS_1, CF_1, FS_1)$ and $TP_2(th) = (VS_2, CF_2, FS_2)$ imply $VS_1 =_L VS_2$.

Proof. First, it should be noted that if the effect type $L$ is derived for $s$, subtyping on statements cannot derive $H$ as the effect type of $s$. The proof of both cases is by induction on the structure of $s$.

1. • $: Immediate.

• $pe$: By the rule TS-PE, the effect type $ℓ^o$ is the effect type of expression $pe$. The execution of $pe$ is composed of one step $\text{run}_{th}$ and the evaluation of $pe$ afterward. The former does not change heap and thread pool. By Lemma 1, the evaluation of $pe$ preserves the low part of heap and the domain of thread pool.

• $if (e_1 == e_2)\{s_1\} else \{s_2\}$: According to the rules TS-If and TS-STUPIDIF, the effect type of expressions $e_1$ and $e_2$ as well as that of statements $s_1$ and $s_2$ is $H$. The execution of $s$ is composed of evaluating the expressions in the condition part and executing the corresponding branch afterward. By Lemma 1, the former does not change the low part of heap and the domain of thread pool. The result holds for the latter by induction hypothesis on $s_1$ and $s_2$.

• $e.f = e'$: By the rule TS-FIELDWRITE, the effect type of $e$ and $e'$ is $H$. Moreover, the security level of the field $f$ in the type (class) of $e$ is $H$. The execution of this statement comprises the evaluation of $e$ and $e'$ as well as the assignment of the value of $e'$ to the field $f$ of the object identified by the value of $e$ afterward. The first and second cases preserve the low part of heap and the domain of thread pool by Lemma 1. In addition, the low part of heap is not changed by such an assignment because only the value of a high field is changed.

• $x = e$: According to the rule TS-VARWRITE, the effect type of $e$ is $H$. To execute this statement, expression $e$ is first evaluated to $v$ which is then assigned to variable $x$. The latter which takes one step, $\text{run}_{th}$, does not change heap and domain of thread pool. By Lemma 1, the evaluation of $e$ does not change the low part of heap and domain of thread pool.

• $\text{return } e$: Similar to the previous case.

• $\{s_1, \ldots, s_n\}$: By induction hypothesis on the sequence $s_1, \ldots, s_n$. 

18
• \((C, \ell) \; x; s_1, \ldots, s_n\): According to the rule TS-Intro, the effect type of \(s_1, \ldots, s_n\) is also \(H\). As the execution of \((C, \ell) \; x;\) takes only one step, \(run_{th}\), which does not change heap and the domain of thread pool, the result holds as induction hypothesis on \(s_1, \ldots, s_n\) implies.

• \(s_1, s_2, \ldots, s_n\) where \(s_1 \neq (C, \ell) \; x;\): According to the rule TS-SEQ, the effect type of \(s_1\) and \(s_2, \ldots, s_n\) is \(H\). The results holds as induction hypotheses on \(s_1\) and \(s_2, \ldots, s_n\) imply.

• \(\text{super}(e_1, \ldots, e_n);\): Similar to the case of \(e'.m(e_1, \ldots, e_n)\) in Lemma 1.

2. It should be noted that by \(VS_1 =_L VS_2\), we mean \(BS_1 =_L BS_2\) for the outermost block scopes associated with the main body of threads.

• \(\cdot\);: Immediate.

• \(pe;\): By the rule TS-PE, the effect type \(\ell^v\) is always set to \(H\). Semantically, the evaluation of \(o.m(v_1, \ldots, v_n)\) or \(\textbf{new} \; (C, \ell)(v_1, \ldots, v_n)\), as the closed frame corresponding to a promotable expression, only changes its dedicated method scope which is removed at the end of the evaluation. Therefore, the block scopes under question are not changed. It can be simply shown that this result is also true of non-promotable expressions. Therefore, the evaluation of \(pe\), as the main part of executing \(pe;\), does not change the block scopes.

• \(\textbf{if} \; (e_1 == e_2) \{\tilde{s}_1\} \; \textbf{else} \{\tilde{s}_2\}\): According to the rules TS-If and TS-STUPIDIf, the effect types of statements \(\tilde{s}_1\) and \(\tilde{s}_2\) have also been derived as \(H\). The execution of \(s\) consists of evaluating the expressions in the condition and executing the corresponding branch afterward. As stated in the previous case, the evaluation of expressions preserves the low view relation on block scopes. Moreover, \(BS_1 =_L BS_2\) is also true as induction hypotheses on \(\tilde{s}_1\) and \(\tilde{s}_2\) imply.

• \(e.f = e';\): By the rule TS-FieldWrite, the effect type of this statement is always set to \(H\). The evaluation of \(e\) and \(e'\) as well as the assignment of a value to a field does not change the block scopes.

• \(x = e;\): According to the rule TS-VarWrite, the security level of variable \(x\) is \(H\). The evaluation of \(e\) does not change the block scopes. Moreover, the low part of the block scope where \(x\) has been declared is not changed by this assignment because this variable is high.

• \(\text{return} \; e;\): The effect type \(\ell^v\) of this statement is always set to \(H\) by the rule TS-RETURN. The evaluation of \(e\) to \(v\) does not change the block scopes. Although the execution of \(\text{return} \; v;\) causes the innermost method scope to be removed, this is not the case for the method scopes associated with the main body of threads. In fact, according to the rule E-MethodVoidStart, \(th\) itself but not \(\text{return} \; th;\) is considered at the end of executing a thread.

• \(\{s_1, \ldots, s_n\}\): By induction hypothesis on the sequence \(s_1, \ldots, s_n\).
• (C, ℓ) x; s₁, . . . , sₙ: According to the rule TS-Intro, the effect type of s₁, . . . , sₙ is H. Moreover, the security level ℓ declared for x is also H. As the execution of (C, ℓ) x; takes only one step, run₁th, which does not change the low part of the block scope where x is declared, the proof is by induction hypothesis on s₁, . . . , sₙ.

• s₁, s₂, . . . , sₙ where s₁ ≠ (C, ℓ) x:: According to the rule TS-Seq, the effect type of s₁ and s₂, . . . , sₙ is H. The proof is by induction hypothesis on s₁ and s₂, . . . , sₙ.

• super(e₁, . . . , eₙ);: Similar to the case of pe;

Lemma 3. If ∆ ⊢ P ok, ∆m(C)(m) = ⊤ −⟨H, ⊤⟩→ ⊤, and (H₁, TP₁, SCFG₁) ⊢ (H₂, TP₂, SCFG₂) appears in an execution of the non-start method m of class C, then H₁ =Ł H₂ and dom(TP₁) = dom(TP₂).

Proof. We consider the two cases below.

• The type of m is not required in its type checking: According to the rules T-ProgDef and T-MBody, the effect type H is derived by the rule T-MDefn. According to this rule, ∅; L; ∅; Γ; ∅ ⊢ ¯s: (κ, τ, ℓ, v, H) |T holds where Γ = x₁ : κ₁, . . . , xn : κₙ, result : (κ, τ, κ, ℓ, H), this : (C, ℓm, H) such that ∆m(C)(m) = ℓm, κ₁, . . . , κₙ −⟨H, ⊤⟩→ κ and mbody(C, m) = (x₁, . . . , xn, ¯s). By Definition 11, an execution of m is a special case of an execution of ¯s. Considering this fact together with the above typing judgment, the result is provided by Lemma 2. Note that in this case, all methods and constructors invoked in m are in the set Ł. Therefore, the claim of the current lemma 3 can be applied to them inductively.

• The type of m is required in its type checking: In this case, the rule T-MBody-Loop is involved. According to the function equalTP, the effect type of m has been derived as H when the same value is considered as the effect type of m in its type checking. As a typing judgment similar to the previous case has been derived here, the result is again provided by Lemma 2. The only difference with the judgment in the previous case is that the set A is not empty. It should be noted that the induction hypothesis can be applied to the invocation of m, although (C, m) cannot be in the set Ł. This is because the type of m at the last round of type checking is the same as its type when it is added to Ł in T-MBody-Loop.

The above discussion is also true of other methods which are in a loop of invocations containing m.

Proof.

Lemma 4. If ∆ ⊢ P ok, ∆c(C) = ⊤ −⟨H, ⊤⟩, and (H₁, TP₁, SCFG₁) ⊢ (H₂, TP₂, SCFG₂) appears in an execution of constructor C, then H₁ =Ł H₂ and dom(TP₁) = dom(TP₂).

Proof.
Proof. The proof is similar to that of Lemma 3. It should be noted that an execution of a constructor is defined just the same as an execution of a method in Definition 11 where \( cnbody \) and \( \Delta_c(C) \) are used instead of \( mbody \) and \( \Delta_m(C)(m) \). \( \square \)

**Lemma 5.** Assume that \( \Delta \vdash P \text{ ok} \) and that either the timing behavior of \( P \) is not nondeducible on high values or there is an access to a low field in \( P \) being deducible on high values. Then, there exists a constructor \( C' \) or a method \( m \) of a class \( C \) such that \( \nu^H \in \mathcal{T}_{C'} \) or \( \nu^H \in \mathcal{T}_m \) where \( \mathcal{T}_{C'} \) and \( \mathcal{T}_m \) are given by \( \Delta_c(C') = \emptyset, (\emptyset, \mathcal{T}_{C'}) \) and \( \Delta_m(C)(m) = \emptyset - (\emptyset, \mathcal{T}_m) \rightarrow \emptyset \).

**Proof.** We consider the following cases.

- The timing behavior of \( P \) is not nondeducible on high values: According to Definition 7, there exist two low-indistinguishable initial global states \( G_1 \) and \( G_2 \) as well as an execution \( E_1 \) under some scheduler \( \sigma \) from \( G_1 \) such that no execution \( E_2 \) under \( \sigma \) from \( G_2 \) satisfies the two conditions in this definition. We show that as \( P \) is well-typed, \( \nu^H \) is derived when the second condition is not satisfied. It is also shown that if the second condition could be satisfied, the first one is satisfied as well. Consider \( E'_1 \) and \( E'_2 \) as prefixes of \( E_1 \) and \( E_2 \) which satisfy both conditions. To show that \( \nu^H \) is derived only due to the denial of the second condition, it is assumed that the other situation resulting in the derivation of \( \nu^H \)—the existence of an access to a low field being deducible on high values—does not occur in \( E'_1 \) and \( E'_2 \). In the proof of the main Theorem 1, it will be shown that the bijection \( \rho \), in the definition of the low-view relation on heaps, exists between the low domains of heaps at which \( E'_1 \) and \( E'_2 \) terminate. Moreover, the values of the same low expressions are shown to be related through \( \rho \). Note that \( \gamma' \) is considered as the bijection between the domains of thread pools at which such prefixes terminate. Consider the first action following \( E'_1 \) and the one following \( E'_2 \). If these actions are equal under \( \gamma' \), there will be a bijection \( \gamma \) between the domains of resulting thread pools. In particular, consider \( insert_{th_1}^{th} \) and \( insert_{th_2}^{th} \) are executed as the result of \( \text{th.start}() \) and \( \text{th'.start}() \). According to TEmethod, any expression \( e \) in \( e\text{.start}() \) is of security level \( L \). Therefore, \( th \) and \( th' \) are related through \( \rho \). In this way, their native types are the same. Therefore, \( \gamma = \gamma' \cup \{(th, th')\} \). If the action following \( E'_1 \) is \( \text{select}_{th} \), the action following \( E'_2 \) can be \( \text{select}_{th'} \) with the condition \( th' = \gamma' \backslash \{(th)\} \). As will be proved in Theorem 1, the scheduling configurations at which \( E'_1 \) and \( E'_2 \) terminate are equal under \( \gamma' \). As \( \sigma \) takes the action \( \text{select}_{th} \), therefore, it can also take the equal action \( \text{select}_{th'} \). If the first action following \( E'_1 \) is \( insert_{th_1}^{th} \) and the corresponding action in \( E_2 \) cannot be \( insert_{th_2}^{th} \) where \( th_2 = \gamma'(th_1) \), then \( \text{th.start}() \) is in \( th_1 \) and \( \text{th'.start}() \) either does not exist in \( th_2 \) or appears in some step taken later. The type system of MMJ does not allow any invocation of \( \text{start} \) in high contexts—the effect type of the code of a high context is \( H \). Therefore, from Lemmas 1-4, no scheduling pass \( \text{insert} \) may appear when the code of a high context...
executes. It should be noted that the only parts of the code that may not be executed, when the difference is limited to the value of high expressions, are those in high contexts. Therefore, \(insert_{th_2}^{th_2}\) appears later in \(E_2\). As low expressions have the same value at the end of \(E_1\) and \(E_2\), only different high values may cause the actions \(insert\) to appear in different steps. In fact, only the statements in high contexts may take different number of run-time steps. In the type system of MMJ, \(t^H\) is derived in the type of a method or constructor which contains some structure leading to a high context that may take different number of steps—see TS-IF and TE-METHOD. When such a method is invoked in another method, the type of the former is included in the type of the latter—see TE-METHOD, TE-NEW, and T-CSUPER. Considering threads, \(w^H\) is derived in the type of a method containing some thread with \(t^H\) in the type of its corresponding run or main—see TE-METHOD and T-MDEFN. If the first action following \(E_1\) is \(terminate_{th_1}\) while the corresponding action in \(E_2\) is not \(terminate_{th_2}\), we can justify the derivation of \(w^H\) similarly.

- There is an access to a low field in \(P\) being deducible on high values: Consider the first step of \(E_1\) or \(E_2\) where one of the closed frames \(o.f = v;\) or \(o'.f = v';\) appears and the other closed frame does not appear in the corresponding step of the other execution. In fact, such closed frames are assumed to be reduced in the same steps before the current one. In the proof of the main Theorem 1, it is proved that the same low expressions have the same value by this step. Therefore, a justification similar to the previous case is also valid here. Note that for the closed frames \(o.f\) and \(o'.f\), it is assumed that both of them appear in \(E_1\) and \(E_2\) even if they are in some high context.

\[\text{Lemma 6.} \] If \(\Delta \vdash P\ ok\) and there are two threads in \(P\) with access to a common low field where one of the accesses is assignment, then there exists a constructor \(C'\) or a method \(m\) of a class \(C\) such that \(\forall o \in T_{C'}\) or \(\forall o \in T_m\) where \(T_{C'}\) and \(T_m\) are given by \(\Delta_{c}(C') = \circ, (\circ, T_{C'})\) and \(\Delta_{m}(C)(m) = \circ \rightarrow (\circ, T_m)\rightarrow \circ\).

**Proof.** According to Definition 13, the closed frame \(o.f = v';\) is in the body of a thread \(th_2\). This closed frame is in fact the result of the reduction of a statement \(e.f = e';\). As \(C_1\)—the native type of \(o\)—is a subclass of the type of \(e\) and the levels of fields do not change in subclasses, the security level of \(f\) in the type (class) of \(e\) is also \(L\). According to the typing rule TS-FIELDWRITE, therefore, \(w\) is derived in the type of run or main corresponding to \(th_2\). If such an assignment occurs in the body of a method or constructor invoked in the body of \(th_2\), \(w\) is again derived through the rules TE-METHOD and TE-NEW. Note that \(w\) has previously been derived in the type of that method or constructor. If \(th_2\) is created through invoking \(\text{start}\), \(w\) is converted to \(w\) as shown in the function \(\text{giveT}\) in TE-METHOD. In fact, \(w\) is in the set \(T\) in the type of the method or constructor containing such an invocation. This is
similarly done if \( t_h_2 \) is the main thread as shown in T-MDEFN. For \( t_h_1 \), \( w \) or \( r \) is also derived in the type of run or main corresponding to this thread—see TE-FIELDACCESS for \( r \). Moreover, it may be converted to \( \omega w \) or \( \omega r \) as explained above. The following cases can be considered for the interaction between \( t_h_1 \) and \( t_h_2 \). For \( j \in \{1, 2\} \),

- \( t_h_j = t_h_{main} \): \( t_h_{3-j} \) may be created in main or another method (constructor) invoked in main. Assume that access to the field in main occurs after the creation of \( t_h_{3-j} \). First, \( \omega w \) or \( \omega r \), associated with \( t_h_{3-j} \), is added to the set \( T' \) when the invocation leading to the creation of \( t_h_{3-j} \) is type checked in the method main. This set is considered as part of typing context in type checking the rest of main. Then, \( w \) or \( r \), associated with the access in main, is also added to this set. As seen in T-MDEFN, therefore, \( \omega \) is finally added to the set \( T_{main} \) through the function conflict. A similar argument holds if \( t_h_{3-j} \) is created after the access to the field in main.

- \( t_h_1, t_h_2 \neq t_h_{main} \) and both threads are created in a unique method (constructor): By invoking start on the first thread \( t_h_j \), \( \omega w \) or \( \omega r \) is added to the set \( T' \) in the rule TE-METHOD. This set is then considered as typing context in type checking the rest of the method (constructor). By the invocation of \( t_h_{3-j} \), \( \omega w \) or \( \omega r \) is returned by the function giveT. Then, \( \omega \) is derived in the set \( T' \) through the function conflict. This set is finally assigned to the set \( T_m \) in the type of that method (constructor).

- \( t_h_1, t_h_2 \neq t_h_{main} \) and threads are created in different methods (constructors): It should be noted that these methods are invoked in main. The symbols \( \omega w \) and \( \omega r \) have previously been derived in the types of these methods. By invoking the first one in main, \( \omega w \) or \( \omega r \) is added to the set \( T' \) in the rule TE-METHOD. This set is then considered as typing context in type checking the rest of main. By the second call, \( \omega w \) or \( \omega r \) is returned by the function giveT. Then, \( \omega \) is derived in the set \( T' \) through the function conflict. The set \( T_{main} \) is finally set to a set containing the members of \( T' \).

\[ \Box \]

**Theorem 1.** If \( \Delta \vdash P \text{ ok} \), then program \( P \) satisfies the noninterference property stated in Definition 9.

**Proof.** We consider the following cases.

- The timing behavior of \( P \) is nondeducible on high values and its accesses to low fields are not deducible on high values: First, we show that every two delete-free executions \( E_1 \) and \( E_2 \) of \( P \) under a high-independent scheduler from two low-indistinguishable initial global states \( G_1 \) and \( G_2 \) are exactly those executions considered in Definition 7. Alternatively, we can show that a typical deterministic scheduler takes the same sequence of passes in every two executions of \( P \) from low-indistinguishable initial
global states. In fact, according to Definition 8, a high-independent scheduler acts in a deterministic manner for $P$ even in the presence of different initial high values. Our proof is by induction on the length of $E_1$ from $G_1$. In fact, we show that under a high-independent scheduler, the sequences of scheduling passes of every execution $E_2$ from $G_2$ are equal to that of $E_1$ under a bijection $\gamma$ with the conditions in Definition 7. As no special requirement is stated for executions $E_2$, we consider an arbitrary execution $E_2$ beginning from $G_2$ in the following proof. Note that it is also proved that the scheduling configurations in the global states at which $E_1$ and $E_2$ terminate are also equal under $\gamma$.

- **Induction Basis:** The initial setting of scheduling in both executions are $lt_1^{\text{init}} = lt_2^{\text{init}} = \{th_{\text{main}}\}$ and $q_1^{\text{init}} = q_2^{\text{init}} = 0$. By $\text{select}_{th_{\text{main}}}$, both executions pass to the same states $ct_1 = ct_2 = th_{\text{main}}$, $lt_1 = lt_2 = \{th_{\text{main}}\}$, and $q_1 = q_2$. The last equality is due to the deterministic behavior assumed for high-independent schedulers.

- **Induction Hypothesis:** The prefixes $E_1'$ and $E_2'$ of the executions $E_1$ and $E_2$ are chosen with equal length. Considering $(H_1',TP_1',S_{CFG}^1)$ and $(H_2',TP_2',S_{CFG}^2)$ as the global states at which $E_1'$ and $E_2'$ terminate, it is assumed that there is a bijection $\gamma'$ from $\text{dom}(TP_1')$ to $\text{dom}(TP_2')$ such that either $th = \gamma'(th)$ or $H_1'(th) = H_2'(th)$ holds for every $th \in \text{dom}(TP_1')$ and the sequences of scheduling passes of transitions in $E_1'$ and $E_2'$ are equal under $\gamma'$.

- **Proof for $E_1$ and $E_2$:** We show that the scheduling passes $S_{CFG}^1 \xrightarrow{a} S_{CFG}^2$, $S_{CFG}^2 \xrightarrow{b} S_{CFG}^2$ of the first transitions of $E_1$ and $E_2$ following $E_1'$ and $E_2'$ are the same under an extended bijection $\gamma$. This is done by considering all scheduling actions. If $S_{CFG}^1 = (ct_1,lt_1,q_1)$ and $S_{CFG}^2 = (ct_2,lt_2,q_2)$, we have $ct_2 = \gamma'(ct_1)$, $lt_2 = \{th_2\}$, $\gamma'(th_1) \wedge th_1 \in lt_1$, and $q_1 = q_2$ by induction hypothesis. The equalities $S_{CFG}^1 = (ct,lt,q)$ and $S_{CFG}^2 = (ct',lt',q')$ are considered in the following cases.

  1. $q_1 = q_2 > 0$ and $a = run_{th}$: Every execution beginning from $G_2$ with the scheduling passes equal to those of transitions in $E_1'$ raises the action $run_{th'}$ where $th' = \gamma'(th)$. In fact, we have $b = run_{th'}$. Otherwise, there is no execution from $G_2$ that satisfies the conditions of Definition 7. Moreover, the equalities $ct = th$, $ct' = th'$, $lt = lt_1$, $lt' = lt_2$, and $q = q' = q_1 - 1$ hold according to the semantic rule for the actions $run_{th}$ and $run_{th'}$. Note that we take $\gamma = \gamma'$.

  2. $q_1 = q_2 > 0$ and $a = insert^{th}_{th_1}$: As justified in the previous case, $b = insert^{th}_{th_2}$ where $th' = \gamma'(th)$. For the same reason, the native types of $th_1$ and $th_2$ are the same. Therefore, we take $\gamma = \gamma' \cup \{\{th_1,th_2\}\}$. Moreover, the equalities $ct = th$, $ct' = th'$, $lt = lt_1 \cup \{th_1\}$, $lt' = lt_2 \cup \{th_2\}$, and $q = q' = q_1 - 1$ hold according to the semantic rule for the pass label $insert$. 

24
* \( q_1 = q_2 > 0 \) and \( a = \text{terminate}_{th} \): As justified above, \( b = \text{terminate}_{th} \), where \( th' = \gamma'(th) \). The equalities \( ct = ct' = th_{nil}, lt = lt_1, lt' = lt_2, \) and \( q = q' = 0 \) hold according to the semantic rule for the pass label \( \text{terminate} \). Note that we take \( \gamma = \gamma' \).

* \( q_1 = q_2 = 0 \) and \( a = \text{select}_{th} \): As high-independent schedulers act in a deterministic manner and the setting in \( S^1_{CFG} \) and \( S^2_{CFG} \) are equal, the corresponding thread \( th' = \gamma'(th) \) is chosen to be executed for the same quantum. In fact, we have \( ct = th, ct' = th', lt = lt_1, lt' = lt_2, q = q' > 0 \). Note that we take \( \gamma = \gamma' \).

Considering \( G^1 = (H_1, TP^1, S_{CFG}) \) and \( G^2 = (H_2, TP^2, S_{CFG}) \), to prove that \( H_1 =_{L} H_2 \) and \( TP^1 =_{L} TP^2 \), we consider changes in low parts of heaps and block scopes affected by the code in the main bodies of threads. The transitions in \( E_1 \) and \( E_2 \) whose labels of scheduling passes are either \( \text{terminate}_{th} \) and \( \text{terminate}_{th} \), or \( \text{select}_{th} \) and \( \text{select}_{th} \), do not change heap or thread pool. Therefore, such transitions can be disregarded in the proof. Moreover, the transitions whose labels of scheduling passes are \( \text{insert}_{th} \) and \( \text{insert}_{th}' \) do not change the heap and variable stack of the threads previously created. They only add the block scopes associated with the threads \( th_1 \) and \( th_2 \). According to TE-METHOD, any expression \( e \) in \( e.\text{start}() \) is of the security level \( L \). By Lemma 1, \( th_1 \) and \( th_2 \) are also of security level \( L \). As we intend to take \( \gamma \) as the bijection \( \mu \) between \( \text{dom}(TP^1) \) and \( \text{dom}(TP^2) \), we show that \( th_2 = \mu(th_1) \) if the level \( \ell_{run} \) in the native type of these identifiers is \( L \). This is required to show that the corresponding block scopes of \( th_1 \) and \( th_2 \) are related through the low-view relation. Note that \( \mu \) is the bijection which is intended to be established between the low domains of \( H_1 \) and \( H_2 \).

For two transitions with labels \( \text{run}_{th} \) and \( \text{run}_{th}' \), the corresponding MJ rules may not be the same. These transitions make changes to heap as well as to the variable stack of the current thread. Therefore, they should be investigated in the proof of soundness. In doing so, we prove the following claims. Note that we intend to construct the bijection \( \mu \). Therefore, we pair off low object identifiers through \( \mu \). Moreover, every pair of closed frames composed of \( o.f = v \); and \( o'.f = v' \); or \( o.f \) and \( o'.f \) are reduced in the same steps of every two executions \( E_1 \) and \( E_2 \) with the conditions in Definition 12.

1. If low object identifiers \( o \in \text{dom}_L(H_1) \) and \( o' \in \text{dom}_L(H_2) \) are the values of the same low expression, every time the statements \( o.f = v \); and \( o'.f = v' \); appear in \( E_1 \) and \( E_2 \) where \( f \) is a low field, the values \( v \) and \( v' \) are connected by \( \mu \).

2. For a low variable \( x \) in the main body of a thread, the reductions of \( x = e \) in \( E_1 \) and \( E_2 \), i.e., \( x = v \); and \( x = v' \);, are such that the values \( v \) and \( v' \) make a pair in \( \mu \).
3. An expression $\text{new } (C, L)(\bar{e})$ in program $P$ is evaluated in both $E_1$ and $E_2$.

4. A statement $(C, L) \ x$; in program $P$, is executed in both $E_1$ and $E_2$.

5. If low object identifiers $o \in \text{dom}_L(H_1)$ and $o' \in \text{dom}_L(H_2)$ are the values of the same low expression, every time the pair of $o.f$ and $o'.f$ or $o.f = v$; and $o'.f = v'$; appears in $E_1$ and $E_2$ where $f$ is a low field, the values of $o.f$ and $o'.f$ are related by $\rho$.

6. The evaluation of a low variable $x$ as a closed frame in $E_1$ and $E_2$ leads to values that make a pair in $\rho$.

We prove these claims by induction on the length of $E_1$. Moreover, $E_2$ represents another given execution. As induction basis, it is immediate that the values of low input parameters (variables) to main are related through $\rho$. This is because initial global states are low-indistinguishable.

As the induction step, for each case, we consider two prefixes $E'_1$ and $E'_2$ of these executions which satisfy all the claims above. Then, we relate low object identifiers $o$ and $o'$ through $\rho$. In this way, the assumption that $o$ and $o'$ are the values of the same low expression in each occurrence of $o.f$ and $o'.f$ as well as $o.f = v$; and $o'.f = v'$; is satisfied by induction hypothesis.

1. Consider $E'_1$ and $E'_2$ terminating at closed frames $o.f = v$; and $o'.f = v'$; These assignments are out of any high context because the statement $e.f = e'$; is out of such contexts according to the type system of MMJ. Moreover, from assumption, such closed frames occur in same steps of $E_1$ and $E_2$. According to the typing rule $\text{TS-FieldWrite}$ for the statement $e.f = e'$; the security level of $e'$ is $L$. It can be easily shown that high variables and fields are not involved in the evaluation of $e'$. By Induction hypothesis, the values of low variables or low fields evaluated are related through $\rho$. Moreover, the values of an expression $\text{new } (C, L)(\bar{e})$ in the two executions are related by $\rho$. Note that even if the values of low fields of the object created by this expression are changed before returning the identifier, their values are related through $\rho$ by induction hypothesis. Therefore, any low expression in $P$, in the two executions, evaluates to values related through $\rho$. In particular, $v$ and $v'$ are related through this bijection. Note that $E'_1$ and $E'_2$ can be considered as terminating at the closed frames considered here. In fact, $e.f = e'$; has been certainly encountered in these prefixes. This is because such an assignment is out of any high context. Moreover, the values of low expressions are also related through $\rho$ as explained above.

2. Consider $E'_1$ and $E'_2$ terminating at closed frames $x = v$; and $x = v'$; The numbers of steps at which $E'_1$ and $E'_2$ terminate may be different. According to the typing rule $\text{TS-VarWrite}$ for the statement $x = e'$; the security level of $e'$ is also $L$. As justified in the previous case, the values $v$ and $v'$ are related through $\rho$ by induction hypothesis.
3. We show that \( E'_1 \) and \( E'_2 \) can be considered as terminating at the closed frame \( \text{new}(C,L)(\bar{e}) \). According to the type system of MMJ, this expression cannot appear in any high context. This is guaranteed by Lemmas 1-4 and the effect type \( H \) for the code of high contexts. However, its evaluation may depend on low values. In fact, it may appear in a conditional structure with low condition or in the body of a method invoked on a low expression. By induction hypothesis, as justified above, the values of any low expression, in the two executions, are related through \( \rho \).

4. Similar to the previous case.

5. Consider \( E'_1 \) and \( E'_2 \) terminating at closed frames \( o.f \) and \( o'.f \). From assumption, these closed frames are in the same steps of \( E_1 \) and \( E_2 \). The value of a field may be changed in two ways: the creation of the associated object through \( \text{new}(C,L)(\bar{v}) \) and the assignment through \( o.f = v \). If there is \( e.f = e' \); which executes in \( E'_1 \) and \( e \) has been evaluated to \( o \), then \( o.f = v \); and \( o'.f = v' \); appear in the same steps of \( E'_1 \) and \( E'_2 \). Moreover, \( v \) and \( v' \) are also related through \( \rho \) by induction hypothesis. The values of \( o.f \) and \( o'.f \) are those \( v \) and \( v' \) that have been assigned to this field through the last occurrence of \( o.f = v \); and \( o'.f = v' \); in \( E'_1 \) and \( E'_2 \). Therefore, these values are related through \( \rho \). If there is no assignment, the values of such closed frames are null which has been assigned when the object is created. Note that the values of some low field \( f' \) of \( o.f \) and \( o'.f \) may have also been changed. If there is a statement \( e_1.f' = e_2 \); and \( e_1 \) has been evaluated to \( o.f \) in \( E'_1 \), then \( e_1 \) has been evaluated to \( o'.f \) in \( E'_2 \). This is because \( e_1 \) is in fact \( e'_1.f \) where \( e'_1 \) has been evaluated to \( o \) and \( o' \) by induction hypothesis. For the same reason, the values of \( e_2 \) in the two executions are also related through \( \rho \).

6. Consider \( E'_1 \) and \( E'_2 \) terminating at the closed frame \( x \). The value of a variable is changed in three ways: \( (C,L)\ x; \ x = v; \) and \( x \) which is a formal parameter that takes the value of its corresponding actual parameter. The case \( x = v; \) is justified in the same way as in the previous case. According to the type system of MMJ, an actual parameter \( e \) is low if its corresponding formal parameter \( x \) is also low. By induction hypothesis, as explained in the first case, the values of \( e \) in the two executions are related through \( \rho \). If \( x \) is not a formal parameter and there is no assignment to it, these values are null by the declaration \( (C,L)\ x; \). There is a special case here. Consider, for example, that \( x \) appears in \( E_1 \) in some step after the step it occurs in \( E_2 \). If an assignment to a field of an object pointed by \( x \) occurs between these steps, they may not be related through \( \rho \) at these steps—although the value of \( x \) is not changed in the two executions. However, if \( E'_2 \) is taken as the prefix terminating at the corresponding assignment to the field in \( E_2 \), the values of \( x \) in the two executions are finally related through \( \rho \).
By the claims above, \( \text{dom}_L(H_1) \) and \( \text{dom}_L(H_2) \) are of equal size and low object identifiers in one can be paired off with a low object identifier in the other having the same native type. In fact, the pair of low object identifiers \( o \) and \( o' \) that are the values of the same low expression \( e \) in each occurrence of \( e.f \) and \( e.f = e' \); are related through \( \rho \). As another result of the claims above, when \( \gamma \) is taken as \( \mu \), the set of low variables in final block scopes of corresponding threads is the same. Moreover, the values of low variables are also related through \( \rho \). We should prove that \( th_2 = \rho(th_1) \) for the corresponding steps \( \text{insert}^{th} \) and \( \text{insert}^{th'} \) in \( E_1 \) and \( E_2 \). As stated earlier, \( e \) in \( \text{e.start()} \) which has been evaluated to \( th_1 \) and \( th_2 \) is of the security level \( L \) according to the type system. As proved above, the values of \( e \) are related through \( \rho \). This completes the proof of the first case.

- Either the timing behavior of \( P \) is not non-deducible on high values or there is an access to a low field in \( P \) being deducible on high values: As proved in Lemma 5, there exists a constructor \( C' \) or a method \( m \) of a class \( C \) such that \( v^H \in T_C \) or \( v^H \in T_m \). According to the typing rule \( \text{T-PROGDEF} \), therefore, the symbol \( \bowtie \) cannot be derived in the type of any constructor or method of well-typed program \( P \). According to Lemma 6, therefore, there cannot be two threads in \( P \) with access to a common low field where one of the accesses is assignment. We can then prove the same claims as the ones in the first case. Although the proofs are similar to the ones given in that case, some differences are explained here.

In the proof of claim (5), closed frames \( o.f \) and \( o'.f \) as well as \( o.f = v; \) and \( o'.f = v' \); in \( E_1' \) and \( E_2' \) are not necessarily reduced in the same steps. It occurs when \( o.f = v; \) and \( o'.f = v' \); are in a thread other than the one containing \( o.f \) and \( o'.f \). For the assignments in the same thread, \( e.f \), with \( e \) evaluated to \( o \) and \( o' \), is reduced after any assignment \( e_1.f = e_2 \); where \( e_1 \) has been evaluated to these object identifiers. This is also true of claim (6) because all assignments to a variable are local to the thread in which that variable is declared. However, note that \( o.f = v; \) and \( o'.f = v' \); cannot be in one thread and \( o.f \) and \( o'.f \) in another thread. This is due to the fact that there cannot be two threads in \( P \) with access to a common low field where one of the accesses is assignment. Apart from this issue, the values \( v \) and \( v' \) are related through \( \rho \) by induction hypothesis. Therefore, the values of \( o.f \) and \( o'.f \) are those assigned to them, in the same thread, through the last occurrence of \( o.f = v; \) and \( o'.f = v' \); in \( E_1' \) and \( E_2' \). Note that there may be only one other thread which changes a low field \( f' \) of \( o.f \) and \( o'.f \). In fact, there may be an assignment \( e_1'.f' = e_2' \); in some other thread. If \( e_1' \) in the form of \( e_1''.f \) has been evaluated to \( o.f \) and \( o.f.f' = v_2 \); occurs in some step before \( o.f \) in \( E_1 \), then we take the prefix \( E_2' \) to the step at which \( o'.f.f' = v_2' \); appears in \( E_2 \) provided it is in some step after \( o'.f \). From induction hypothesis, this step definitely occurs. Moreover, \( v_2 \) and \( v_2' \) are related through \( \rho \). This is similarly applied to claims (1), (2), and (6).
Finally, for claim (1), although $v$ and $v'$ in $o.f = v$ and $o'.f = v'$ are related through $\rho$, the field $f$ may also be changed after these steps in only one of the executions $E_1$ or $E_2$, and the resulting value may not be related to $v$ or $v'$. In fact, the corresponding modification of $f$ may be before these steps in the other execution. This occurs only if these assignments are in a thread other than the one containing $o.f = v$ and $o'.f = v'$. However, this cannot be the case due to the fact that there cannot be two threads in $P$ with access to a common low field where one of the accesses is assignment.