Online Appendix to:

Static Checking for Multiple Start of Threads in a Type-Safe Multithreaded Java

Zeinab Iranmanesh\textsuperscript{a} \quad Mehran S. Fallah\textsuperscript{ab}

\textsuperscript{a}. Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran
\textsuperscript{b}. School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

Contents

A The Type System of MTMJ
   A.1 Required Rules for Recursive Methods and Constructors

B Typing Frame Stacks
   B.1 The Function \textit{subs}

C Proofs
   C.1 Lemmas
   C.2 Progress
   C.3 Preservation
A The Type System of MTMJ

The typing rules of MTMJ for expressions are given in Fig. 1. The judgment $\Delta \vdash \Delta \text{ok}$ states that the class table $\Delta$ is well-formed. That is, the types of fields, methods, and constructors obtained from $\Delta$ are valid in the sense that they comply with $\Delta$. For example, the fields of a class $C$ are well-formed according to the rule

$$
\frac{\text{dom}(\Delta_f(C)) = \{f_1, \ldots, f_n\}}{
\Delta \vdash \Delta_f(C)(f_1) \text{ ok} \quad \ldots \quad \Delta \vdash \Delta_f(C)(f_n) \text{ ok}}{
\Delta \vdash \Delta_f(C) \text{ ok}}
$$

where the premise $\Delta \vdash \Delta_f(C)(f_i) \text{ ok}$ is defined by

$$
C \in \text{dom}(\Delta).
$$

Similarly, $\Delta \vdash \Gamma \text{ ok}$ means that the types in $\Gamma$—a mapping from variables to their types—are valid.

The typing rules of MJ for nonpromotable expressions can be easily extended into MTMJ; the rules $\text{TE-Var}$ and $\text{TE-Null}$ simply copy $\Phi$ to the right side of the judgment. The rules for casting, e.g., $\text{TE-UpCast}$, and the rule $\text{TE-FieldAccess}$ copy the set $\Phi'$ from their premises into their conclusions. It is worthy of mention that although null pointers, such as $null.f$, are accepted by the type system, they lead to run-time exceptions. In fact, we do not concentrate on preventing this kind of error statically.

Consider the cases like $(C)(C')e$ where $e$ is of type $D$ such that $D \prec C'$, $C \prec C'$, and there is no subclass relation between $C$ and $D$. Such an expression is well-typed in Java, although it leads to casting a $D$ object to a $C$ one at run-time. Thus, to achieve type soundness, the latter kind of casts should be typable through the rule $\text{TE-StupidCast}$—conditional structures whose condition part contains two expressions whose types have no subclass relation should be typable for the same reason. In order to emphasize that valid programs should not contain such constructs, separate rules are considered.

The rules for casting have a new premise $\neg \Delta_{th}(C)$ in comparison with MJ, where $C$ is the type of the argument of casting which is the expression $e$ here. Since $(C')e$ is an alias for $e$, $e$ should not be a thread or contain a thread in its fields. We enforce such a restriction by using $\Delta_{th}(C)$ which is false when no thread appears in $e$. To be type-safe, MTMJ also requires $\Delta_{th}$ to examine the subclasses of its argument. This is because the class—type—of the value of an expression may be a subclass of the static type of that expression. In the rule $\text{TE-StupidCast}$, the premise $\neg \Delta_{th}(C')$ prevents casting to a class with threads when $e$ is not of type $\text{Thread}$. This premise is not included in $\text{TE-DownCast}$ because it is implied by the other premise $\neg \Delta_{th}(C)$. Although this is not the case for $\text{TE-UpCast}$, only the third condition of $\Delta_{th}$ may cause this function to be true for $C'$ when $\neg \Delta_{th}(C)$ is satisfied. Therefore, $(C')e$ cannot be a thread because there is no way for assigning a thread value to such a casting expression.

The typing rules of MTMJ for statements are given in Fig. 2. The rule $\text{TS-NoOp}$ copies $\Phi$ from the hypotheses of the typing judgment into its consequent. The rules $\text{TS-Return}$ and $\text{TS-Block}$ simply repeat $\Phi'$ of their premises in their conclusion.

Figure 3 shows typing rules for the $\text{super}$ call. Moreover, the rules given in Fig. 4 type check a program.
Figure 1 – Typing rules for expressions. The symbol “↑” stands for undefined and “↓” for defined.
⊢ Δ \vdash \Gamma \vdash \Phi \vdash \Phi' \quad \text{(TS-NoOp)}

\Delta; \Gamma; \Phi \vdash \Phi' \quad \text{(TS-PE)}

\Delta; \Gamma; \Phi \vdash e_1 : C_1 | \Phi_1 \quad \Delta; \Gamma; \Phi \vdash e_2 : C_2 | \Phi_2
\Delta; \Gamma; \Phi_2 \vdash \bar{s}_1 : \text{void} | \Phi_3 \quad \Delta; \Gamma; \Phi_2 \vdash \bar{s}_2 : \text{void} | \Phi_4
C_1 \subseteq C_2 \lor C_2 \subseteq C_1 \quad \Phi' = \Phi_3 \cup \Phi_4

\Delta; \Gamma; \Phi \vdash \text{if} (e_1 == e_2) \{ \bar{s}_1 \} \text{ else } \{ \bar{s}_2 \} : \text{void} | \Phi'
\quad \text{(TS-If)}

\Delta; \Gamma; \Phi \vdash e_1 : C_1 | \Phi' \quad \Delta; \Gamma; \Phi \vdash e : C_2 | \Phi''
\Delta_f (C_1)(f) = C_3 \quad C_2 \subseteq C_3
\quad \frac{e = \text{null} \lor e = \text{new } C_2(e_1, \ldots, e_n) \lor \neg \Delta_{th}(C_2)}{\Delta; \Gamma; \Phi \vdash e.f = e : \text{void} | \Phi''}
\quad \text{(TS-FieldWrite)}

\Delta; \Gamma; \Phi \vdash x : C_3 | \Phi' \quad \Delta; \Gamma; \Phi \vdash e : C_2 | \Phi''
\quad \frac{C_2 \subseteq C_3 \quad x \neq \text{this}}{\Delta; \Gamma; \Phi \vdash x = e : \text{void} | \Phi''}
\quad \text{(TS-VarWrite)}

\Delta; \Gamma; \Phi \vdash e : C | \Phi'
\Delta; \Gamma; \Phi \vdash \text{return } e : C | \Phi'
\quad \text{(TS-Return)}

\Delta; \Gamma; \Phi \vdash \{ s_1 \ldots s_n \} : \text{void} | \Phi'
\Delta; \Gamma; \Phi \vdash \{ s_1 \ldots s_n \} : \text{void} | \Phi'
\quad \text{(TS-Block)}

\Delta; \Gamma; \Phi \vdash e : C_1 : \text{void} | \Phi'
\quad \frac{s_1 \neq C_1 \quad \Delta; \Gamma; \Phi \vdash \{ s_1 \ldots s_n \} : \text{void} | \Phi'}}
\quad \text{(TS-Intro)}

\Delta; \Gamma; \Phi \vdash s_1 : \text{void} | \Phi_1 \quad \Delta; \Gamma; \Phi \vdash s_2 \ldots s_n : \tau | \Phi'
\Delta; \Gamma; \Phi \vdash \{ s_1 \ldots s_n \} : \tau | \Phi'
\quad \text{(TS-Seq)}

Figure 2 - Typing rules for statements.
A.1 Required Rules for Recursive Methods and Constructors

The judgment \( \Delta; (C_1, m_1), \ldots, (C_n, m_n) \vdash loop(C, m) \) defined by MethodLoop in Fig. 5 states that method \( m \) of class \( C \) is a member of some loop. This is derived by investigating the body of \( m \) through LoopDet. The sequence \( (C_1, m_1), (C_2, m_2), \ldots, (C_n, m_n) \) in LoopDet is \( (C_0, m_0) \) when this rule is initially applied through MethodLoop and, in turn, through TE-Method. A pair \( (C_{n+1}, m_{n+1}) \) or \( (C_{n+1}, C_{n+1}) \) is appended to this sequence if method \( m_{n+1} \) of class \( C_{n+1} \) or the constructor 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 method and constructor invocations in \( \bar{s} \) are taken into account. For an invocation, a loop is detected if the method or constructor invoked is \( m_1 \) of \( C_1 \). It should be noted that the judgment \( \Delta; \Gamma \vdash e : C \) is used to show that the MJ type of \( e \) is \( C \). This judgment is the same as the typing judgment of MTMJ except that the components other than those concerning native types are removed.

B Typing Frame Stacks

MTMJ extends the judgment \( \Delta; H, VS \vdash FS : \tau \rightarrow \tau' \) of MJ with \( \Phi \) in its hypotheses. Thus, as shown in Fig. 7, it also extends the rules defining such a judgment—we do not repeat those rules that simply add only a \( \Phi \) to the hypotheses of the judgments appearing in the corresponding rule of MJ. To type check a frame stack \( FS \) comprising several closed or open frames, we use MTMJ’s typing rules for expressions and statements. Nevertheless, the typing rules should be adapted for type checking a frame containing object identifiers and holes. The rules TE-OID and TE-Hole in Fig. 6 are for type checking an object identifier and a hole, respectively. Here, the type of a hole in the typing context is the type of the value replacing that hole.

The typing rules of MTMJ, however, may reject a healthy frame because object identifiers and holes do not appear in the rules. For example, if \( C \) is the type of "•"
\[
\text{if } m = \text{start then}
\]
\[
\left( \begin{array}{c}
C \xrightarrow{\text{Thread}} \land C \neq \text{Thread} \\
P = * \text{ class } C \text{ extends } C' \{ * \text{ and } m_1 \ldots m_n \} *
\end{array} \right)
\]
\[
\Rightarrow \forall i \leq n. \ m_i \neq \text{ void start}(i) \star
\]
\[
m_g = \text{giveMethod}(C, m)
\]
\[
\Delta_{m}(C)(m_g) = C_1, \ldots, C_n \to \tau
\]
\[
\text{if } \tau \neq \text{ void then } \Delta_{m}(\tau)
\]
\[
\text{if } \Delta_{c}(C, m_g) \vdash \text{loop}(C, m_g)
\]
\[
\text{then } \Delta_1 = \Delta(\Delta_{m}(C)(m_g) \rightarrow C_1, \ldots, C_n \to \tau) \text{ else } \Delta_1 = \Delta
\]
\[
\text{mbody}(C, m_g) = (x_1, \ldots, x_n, s)
\]
\[
\Gamma = x_1: C_1, \ldots, x_n: C_n, \text{ this } : C
\]
\[
\Delta_1; \Gamma; \emptyset \vdash s : \tau|\Phi'
\]
\[
\Delta' = \Delta(\Delta_{m}(C)(m_g) \rightarrow C_1, \ldots, C_n \to \tau)
\]
\[
\Delta \vdash \text{mbody}(C, m) \text{ ok}|\Delta'
\]  

(T-MDEFN)

\[
\text{dom}(\Delta_{m}(C)) = \{ m_1, \ldots, m_n \} \quad \Delta_0 = \Delta
\]
\[
\Delta_{n} \vdash \text{mbody}(C, m_{n+1}) \text{ ok}|\Delta_{n+1} \quad \Delta_{n+1} \vdash \text{mbody}(C, m_{n+1}) \text{ ok}|\Delta_{n+1}
\]
\[
\Delta \vdash \text{C mok}|\Delta'
\]  

(T-MBODIES)

\[
\Delta_{c}(C) = C_1, \ldots, C_n *
\]
\[
\text{if } \Delta_{c}(C, C) \vdash \text{loop}(C, C)
\]
\[
\text{then } \Delta_1 = \Delta(\Delta_{c}(C) \rightarrow C_1, \ldots, C_n, \emptyset) \text{ else } \Delta_1 = \Delta
\]
\[
\text{cmbody}(C) = (x_1, \ldots, x_n, \text{ super}(\bar{e}_1', \ldots, \bar{e}_n'); s)
\]
\[
\Gamma = x_1: C_1, \ldots, x_n: C_n, \text{ this } : C
\]
\[
\Delta_1; \Gamma; \emptyset \vdash \text{super}(\bar{e}_1', \ldots, \bar{e}_n'); \text{ void }|\Phi
\]
\[
\Delta_1; \Gamma; \emptyset \vdash s : \text{ void }|\Phi
\]
\[
\text{if } \Delta_{m}(C) \text{ then } \Phi' = \emptyset
\]
\[
\Delta' = \Delta(\Delta_{c}(C) \rightarrow C_1, \ldots, C_n, \Phi')
\]
\[
\Delta \vdash \text{C cok}|\Delta'
\]  

(T-CDEFN)

\[
\text{dom}(\Delta) = \{ C_1, \ldots, C_n \} \quad \Delta_0 = \Delta
\]
\[
\Delta_0 \vdash C_1 \text{ cok}|\Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash C_n \text{ cok}|\Delta_n
\]
\[
\Delta_n \vdash C_1 \text{ mok}|\Delta_{n+1} \quad \ldots \quad \Delta_{2n-1} \vdash C_n \text{ mok}|\Delta_{2n}
\]
\[
\text{dom}(\Delta_{m}(C)) = \{ m_1, \ldots, m_n \} \quad \{ t = 1, \ldots, n \}
\]
\[
\forall i \leq n. \ i \leq j \leq k_C, \text{ chTP}(\Delta_{m}(m_j), C_i)
\]
\[
P = *; s \quad \Delta_{2n}; \emptyset; \emptyset \vdash s : \text{ void }|\Phi
\]
\[
\Delta \vdash P \text{ ok}
\]  

(T-PROGDEF)

Figure 4 – Typing programs.
\[
\text{mbody}(C, m) = (x_1, \ldots, x_j, \bar{s})
\]
\[
\Delta_m(C)(m) = C_1, \ldots, C_j \rightarrow \tau
\]
\[
\Gamma = x_1 : C_1, \ldots, x_j : C_j, \text{this} : C
\]
\[
\Delta; \Gamma; (C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}(\bar{s})
\]
\[
\Delta; (C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}(C, m)
\]
\[
\text{(MethodLoop)}
\]
\[
\text{cnbody}(C) = (x_1, \ldots, x_j, \bar{s})
\]
\[
\Delta_n(C) = C_1, \ldots, C_j, \ast
\]
\[
\Gamma = x_1 : C_1, \ldots, x_j : C_j, \text{this} : C
\]
\[
\Delta; \Gamma; (C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}(\bar{s})
\]
\[
\Delta; (C_1, m_1), \ldots, (C_n, m_n) \vdash \text{loop}(C, C)
\]
\[
\text{(ConsLoop)}
\]
\[
\forall C', m'.
\]
\[
\bar{s} = \ast C_1' \cdot x_1'; \ldots \cdot * C_k' \cdot x_k'; \ast \cdot e.m_1'(\bar{e}) \ast \land
\]
\[
\Delta; \Gamma, x_1' : C_1', \ldots, x_k' : C_k', \ast \cdot e ; C' \land m' = \text{giveMethod}(C', m_1') \lor
\]
\[
(\bar{s} = \ast \text{new} C'(\bar{e}) \ast \land m' = C') \lor
\]
\[
(\bar{s} = \ast \text{super}(\bar{e}) \ast \land \Gamma = \ast, \text{this} : C_1' \land C_1' \prec_1 C' \land m' = C')
\]
\[
\Delta; (C_1, m_1), \ldots, (C_n, m_n), (C', m') \vdash \text{loop}(C', m')
\]
\[
\Delta; \Gamma; (C_1, m_1), (C_2, m_2), \ldots, (C_n, m_n) \vdash \text{loop}(\bar{s})
\]
\[
\text{(LoopDet)}
\]

\[\textbf{Figure 5} - \text{Some judgments that appear in the typing rules. The symbol "\ast" stands for wild card.}\]

\[
\Gamma = \ast, o : C, \ast \vdash \Delta \text{ ok} \quad \Delta \vdash \Gamma \text{ ok}
\]
\[
\Delta; \Gamma; \Phi \vdash o : C\Phi
\]
\[
\text{(TE-OID)}
\]
\[
\Gamma = \ast, \ast : C, \ast \vdash \Delta \text{ ok} \quad \Delta \vdash \Gamma \text{ ok}
\]
\[
\Delta; \Gamma; \Phi \vdash \ast : C\Phi
\]
\[
\text{(TE-Hole)}
\]

\[\textbf{Figure 6} - \text{Typing rules for object identifiers and "\ast". The symbol "\ast" stands for wild card.}\]
The reason is that the closed frame return saved from wrong rejections. In general, the value of an expression expected by the rule. To resolve this, we define a function and ∆.

\[ \Delta, context(H, MS \circ VS); \Phi + e : \tau | \Phi' \]

\[ FS' = \text{subs}(\text{return } e; FS, \Delta, H, MS \circ VS) \]

\[ \Delta, VS, \Phi' \vdash FS' : \tau \rightarrow \tau' \]

\[ \Delta, H, MS \circ VS, \Phi \vdash (\text{return } e) \circ FS : \tau'' \rightarrow \tau' \]

(TF-StackMethod2)

\[ \text{OF} \neq (\text{return } \bullet) \]

\[ \Delta, context(H, VS); \bullet :: C; \Phi + OF : \tau | \Phi' \]

\[ FS' = \text{subs}(\text{OF}, FS, \Delta, H, VS) \]

\[ \text{if } (FS \neq FS') \text{ then } \tau'' = C \text{ else } \tau'' = \tau \]

\[ \Delta, VS, \Phi' \vdash FS' : \tau'' \rightarrow \tau' \]

(TF-StackOpen)

\[ CF \neq (\text{return } e); CF \neq s_1, \ldots, s_n \land n > 1 \]

\[ CF \neq (\{\}) \text{ CF } \neq C \times \text{ CF'} = \text{CF} \]

\[ \text{if } CF = (x = \alpha) \text{ then } CF' = \text{subs}(o, x = \bullet; o, \Delta, H, VS) \circ [] \]

\[ \text{if } CF = (v, f = \alpha) \text{ then } CF' = \text{subs}(o, v, f = \bullet; o, \Delta, H, VS) \circ [] \]

\[ \Delta, context(H, VS); \Phi + CF' : \tau | \Phi' \]

\[ FS' = \text{subs}(CF, FS, \Delta, H, VS) \]

\[ \Delta, VS, \Phi' \vdash FS' : \tau \rightarrow \tau' \]

(TF-StackClosed)

Figure 7 - The rules for type checking a frame stack.

and \( \Delta_{th}(C) \) is true, the type of “\( x = \bullet \)” cannot be derived by the rule TS-VarWrite. The reason is that TS-VarWrite expects an expression of the form “\textbf{new } C(\bar{e})” and not “\( \bullet \).” Nevertheless, the reductions of dynamic semantics will replace the hole with the value of an expression expected by the rule. To resolve this, we define a function \text{subs} which replaces holes with appropriate expressions so that healthy frames can be saved from wrong rejections. In general, \text{subs} takes a frame \( F \) and a frame stack \( FS \) together with \( \Delta, H \), and \( VS \). It then utilizes \( F \) to replace the hole in an open frame of \( FS \) by an appropriate expression. The details of the function \text{subs} can be found in Section B.1.

The rule TF-StackMethod2 in Fig. 7 type checks a frame stack composed of the closed frame “\text{return } e;” and the frame stack \( FS \). First, \( e \) is type checked against \( \Delta, \Phi \), and the typing context obtained from \text{context}(H, MS \circ VS). It may also be required to change \( FS \) on the basis of “\text{return } e;” by applying \text{subs}. The frame stack \( FS' \) is the result of such a change which is type checked against the set \( \Phi' \) obtained from type checking \( e \). Note that for type checking \( FS' \), the last method scope \( MS \) is removed from the variable stack. This is because the closed frame “\text{return } e;” signifies that the current method terminates, and therefore, the expressions in \( FS' \) do not have access to the local variables of this method. It is worth mentioning that for the open frame “\text{return } \bullet,” the typing rule TF-StackMethod of MJ can be used without change except that \( \Phi \) is added to its typing judgments.

The rule TF-StackOpen is for type checking the composition \( OF \circ FS \), where \( OF \) is an open frame. The typing rules of MTMJ are first utilized to type check \( OF \). In doing so, the type of “\( \bullet \)” is set to \( C \). This is due to the input type of \( OF \circ FS \) which shows that the hole of \( OF \) will be replaced with an expression of type \( C \). For a valid type checking, \( OF \) and \( FS \) are also passed to \text{subs}. It returns the frame stack \( FS' \) which is type checked against the set \( \Phi' \) obtained from type checking \( OF \). Note that \text{subs} changes \( FS \) only if \( OF \) is of the form “\textbf{new } C(\bullet)” in which case the first
hole of $FS$ is replaced by $OF$. Thus, the hole of $OF$ is now the first hole of $FS'$. This implies that the input type of $FS'$ is $C$. If $FS$ is not changed by $subs$, the input type of $FS$ is set to $\tau$, the type of $OF$, because $OF$ is evaluated as the input to $FS$.

For the composition of a closed frame $CF$ and a frame stack $FS$, the typing rule $TF$-$StackClosed$ is presented. Note that this rule is not applied to a number of closed frames stated in the first two lines of the premises of the rule. For the closed frames not excluded from the rule, $CF'$ is set to $CF$ unless $CF$ is “$x = o$” or “$v.f = o$”. For these two cases, the function $subs$ is invoked to check if $\Delta$ is true for the type of the object identifier “$o$” and if “$o$” is the value of an expression of the form “new $C(e)$”. Further, the type derived for $CF'$ and for the input type of $FS'$ should be the same even if $FS$ is changed by $subs$. The arbitrary type $\tau'$ in the conclusion indicates that if a closed frame appears as the first frame of a frame stack, the input type of that frame stack can be any type.

B.1 The Function $subs$

The function $subs$, given in Fig. 8, replaces the hole of an open frame in a frame stack $FS$ with an appropriate expression. In particular, $subs$ makes the frames “$x = o$” and “$v.f = o$” typable only if they fulfill the conditions of the typing rules $TS$-$VarWrite$ and $TS$-$FieldWrite$, respectively. According to these rules, “$x = e$” and “$v.f = e$” are typable only if $e$ is of the form new $C(e)$—here, $e$ is not null and $\Delta$ is true for its type. If $F$, which is the frame preceding $FS$ and its value will be substituted for $\ast$, is of the form new $C(\ast)$, then $subs$ replaces $\ast$ with $F$. By $\ast$ as a parameter, we mean it may be $\ast$, a value, or an expression that has not been evaluated yet. Such an $F$ appears during the evaluation of the parameters of a constructor $C$.

In the execution of the body of a constructor $C$, “return $o$” is the last frame to be executed. For this frame, $\bullet$ is replaced by new $C(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ represent the values of the parameters to the constructor. These values are in the last method scope $MS$ and are obtained from the function $eval$. Note that, by the condition $\Delta(C)$, “return $o$” can only be resulted from the execution of a constructor. At the end of executing the body of constructor, “return $o$” reduces to “$o$”. In this case, $subs$ replaces $\bullet$ with new $C(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ can be the values of the parameters to the constructor that are now in $H$. These values cannot be accessed through $eval$ because the corresponding method scope is removed when the execution of the constructor $C$ terminates. Note that “$o$” may be the value of a local variable or field. To ensure that this is not the case, the values in $H$ and the current $MS$ are checked.

The expression substituted for $\bullet$ is typable. In particular, the types of actual parameters comply with the types of the corresponding formal parameters. Moreover, such an expression does not modify the set $\Phi$ as part of the context for type checking the remaining frame stack $FS'$. This is because no invocation of start appears in the parameters and the body of the constructor $C$.

For the frame “$\bullet$ start()”, $subs$ replaces $\bullet$ with the expression that will be substituted for it in the reductions defined by the dynamic semantics. In this way, “$\bullet$ start()” is typable only if start has not yet been invoked on that expression. The function $subs$ deals with the expressions of the form (1) or (2) stated in Lemma 1. The same holds for “return $o$”, as the last statement resulted from invoking a constructor where the returned value “$o$” is substituted for $\bullet$. Note that “$\bullet$ start()” may appear in $FS$ explicitly or may be the result of replacing a formal parameter by $\bullet$ as the corresponding actual parameter. For instance, “new $C(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)$”
Figure 8 - The function \( \text{subs} \) that replaces a hole of a frame with an appropriate expression. We use \((\alpha_j)_{j=1}^{k}\) for the sequence \(\alpha_1 \alpha_2 \ldots \alpha_k\) if \(k \geq 1\) and an empty string if \(k = 0\). Moreover, \(S_1, \ldots, S_6\) are the logical statements in which \(H_T(o)\) and \(H_V(o)\) are the type and field values of \(o\)'s in \(H\). Furthermore, the function \(\text{eval}(MS, x)\) returns the value of \(x\) in a method scope \(MS\).
is a frame in which the parameters preceding \(e_i\) have been evaluated, \(e_i\) is now being evaluated, and the subsequent parameters have not been evaluated yet. The function \(\text{subs}\) modifies this frame only if \(x_i\), the formal parameter corresponding to \(e_i\), is in \(\Phi_C\). This means that there is an invocation of \texttt{start} on \(x_i\) in the body of the constructor \(C\).

C Proofs

C.1 Lemmas

\textbf{Lemma 1.} Any well-typed expression \(e\) of type \(C \prec \text{Thread}\) is of one of the following forms.

1. A local variable \(x\).
2. \(x.f_1.f_2.\cdots.f_n\), i.e., a local variable \(x\) followed by a sequence of field references.
3. \texttt{new} \(C(e)\).
4. \texttt{new} \(C'(e).f_1.f_2.\cdots.f_n\).

\textit{Proof.} We prove that other forms of \(e\) cannot be of the type stated in Lemma 1. In doing so, we consider the following cases.

- \(e = (C')e':\) The type of this expression is \(C'\) and we should have \(C' \prec \text{Thread}\). Thus, \(\Delta_{th}(C')\) is true. The typing rules for casting have \(\neg \Delta_{th}(C)\) as a premise where \(C\) is the type of \(e'\). As \(\Delta_{th}(C)\) is true for the cases \(C \prec C'\) and \(C' \prec C\), however, the typing rules for casting cannot be applied and \((C')e'\) would not be typable.

- \(e = (C')e'.f:\) The type of this expression is \(C'' = \Delta_f(C')(f)\) and \(C'' \prec \text{Thread}\). Therefore, the class \(C'\) has a field \(f\) whose type is \text{Thread}. This implies that \(\Delta_{th}(C')\) is true. Moreover, for both cases \(C \prec C'\) and \(C' \prec C\), \(\Delta_{th}(C)\) is true in which \(C\) is the type of \(e'\). Note that \(f\) is also a field in the subclasses of \(C'\). Therefore, \((C')e'\) is not typable according to the typing rules for casting.

- \(e = e'.m(e_1,\ldots,e_n):\) The type of this expression is \(\tau\), the return type of \(m\). Moreover, \(\tau \prec \text{Thread}\). Therefore, \(\Delta_{th}(\tau)\) is true. According to \text{TE-METHOD}, however, \(\Delta_{th}(\tau)\) should be false. Hence, \(e'.m(e_1,\ldots,e_n)\) is not typable.

- \(e = e'.m(e_1,\ldots,e_n).f:\) The type of this expression is \(C' = \Delta_f(\tau)(f)\) where \(\tau\) is the return type of \(m\). Since the condition \(C' \prec \text{Thread}\) should be satisfied, the class \(\tau\) has a field \(f\) whose type is \text{Thread}. Therefore, \(\Delta_{th}(\tau)\) is true which implies that \text{TE-METHOD} cannot be applied here.

- The above forms followed by a sequence of fields rather than a single field: Such expressions, e.g., \((C')e'.f_1.f_2.\cdots.f_n\) or \(e'.m(e_1,\ldots,e_n).f_1.f_2.\cdots.f_n\) are also ill-typed. This is due to the fact that \(\Delta_{th}\) returns true for a class \(C\) with a field of type \text{Thread} buried under a sequence of fields.

\[\Box\]

\textbf{Lemma 2.} Every expression \(e\) evaluating to a value in \(\text{dom}(TP)\) is of the forms stated in Lemma 1.
Proof. The only way for a value \( th \) to be added to \( \text{dom}(TP) \) is the invocation of \texttt{start} on \( th \). This is realized by the rule \texttt{E-MethodVoidStart}. Moreover, the expression \( e \) is evaluated to \( th \) through the rule \texttt{E-Translate}. According to \texttt{E-MethodVoidStart}, the type of \( th \) is a subclass of \texttt{Thread}. It can be shown that the type of a value is a subclass of the type of the expression evaluating to that value (Preservation). Therefore, the type of \( e \) is a subclass of \texttt{Thread} and this expressions is of the forms stated in Lemma 1. Note that an expression of the forms excluded from Lemma 1 cannot be an alias for an expression of the forms stated there. In effect, the statements leading to such an alias are ill-typed. More precisely, the type of those expressions excluded from Lemma 1 cannot be a subclass of \texttt{Thread}, though it is required for the application of the rules \texttt{TS-VarWrite} and \texttt{TS-FieldWrite}. For an expression of the form (3), it should be noted that a different value is returned each time \texttt{new} \( C(e) \) is invoked. Therefore, the value that may be assigned to an expression of the forms excluded from Lemma 1 is different from the value on which \texttt{start} is invoked. Finally, \( \Delta_{th} \) may not be true for the return type of a method or the type of an argument of casting. In this way, other kinds of aliasing are also prohibited. \( \square \)

**Lemma 3.** A well-typed expression \( e \) of type \( C \prec \text{Thread} \) cannot be an alias for any well-typed \( e' \) of type \( C' \prec \text{Thread} \).

Proof. The expressions \( e \) and \( e' \) are of the forms stated in Lemma 1. We show that the type system of MTMJ does not type check a program whose execution results in the same value for the two expressions \( e \) and \( e' \). The assignment of \( e \) to \( e' \) is type checked by \texttt{TS-VarWrite} or \texttt{TS-FieldWrite}. If \( e \) is \( x \), \( \Delta_{th}(C) \) is true where \( C \) is the type of \( x \). Similarly, \( \Delta_{th}(C) \) is true if \( e \) is \( x.f_1.f_2.\cdots.f_n \) and \( C \) is the type of \( f_n \). Since \( e \) is not of the form \texttt{new} \( C(e) \) and \( \Delta_{th} \) is true for its type, the assignment of \( e \) to \( e' \) is not type checked by the two rules above. This is also the case for the assignment of \( e' \) to \( e \). The above argument can also be applied to the case where expressions are of the form \texttt{new} \( C(e).f_1.f_2.\cdots.f_n \). Note that although \texttt{new} \( C(e) \) can be assigned to an expression in the rules \texttt{TS-VarWrite} and \texttt{TS-FieldWrite}, the new object identifier returned is assigned to the left side of the assignment such that the corresponding object can only be accessed through the expression in the left. Moreover, \texttt{this} may be assigned to \( e' \) in the body of the constructor of \( C \) such that \( e \) becomes an alias for \( e' \) when an expression of the form \texttt{new} \( C(e) \) is assigned to \( e \). Such an assignment is not typeable either, as it is also checked by the rules above. \( \square \)

**Corollary 1.** There is at most one expression of the form (1) or (2)—stated in Lemma 1—in the program text evaluating to a given \( th \in \text{dom}(TP) \).

Proof. From Lemma 2, an expression of the form (1) or (2)—stated in Lemma 1—whose value is in \( \text{dom}(TP) \) is of type \( C \prec \text{Thread} \). Lemma 3 states that there is no alias for such an expression. \( \square \)

**Lemma 4.** Every expression of the form (3) or (4) in Lemma 1 evaluates to a thread identifier that is not in \( \text{dom}(TP) \).

Proof. The type of \texttt{new} \( C(e) \) is \( C \) which should be a subclass of \texttt{Thread}. In this way, \( \Delta_{th}(C) \) is true. Moreover, the type of \texttt{new} \( C(e).f_1.f_2.\cdots.f_n \) is \( C' \) which should also be a subclass of \texttt{Thread}. According to the definition of \( \Delta_{th} \), \( \Delta_{th}(C) \) is true for \texttt{new} \( C(e).f_1.f_2.\cdots.f_n \) as well. The new thread identifier \( th \) returned by an expression of the forms above may not be started in the body of the constructor of \( C \). This is because no invocation of \texttt{start} is allowed in the constructor of a class \( C \).
for which $\Delta_{th}(C)$ is true, a premise in the rules TE-New and T-Defn. Since the object corresponding to $th$ is not created when $\bar{e}$ is evaluated, it cannot be started during the evaluation of $\bar{e}$. Moreover, $th$ may not be the value of another expression on which $\texttt{start}$ has been invoked prior to the invocation of the constructor. In effect, from Lemma 3 and Corollary 1, such an expression and $\texttt{this}$ or $\texttt{this}.f_1.f_2.\cdots.f_n$ do not make aliases for the same object.

\[ \text{Proof.} \]

We show that an MTMJ transition exists for each well-typed nonterminal 

\[ \text{start} \]

during the evaluation of $\bar{e}$ object corresponding to $th$ from Lemma 3 and Corollary 1, such an expression and this or this.$f_1.f_2.\cdots.f_n$ do not make aliases for the same object.

\[ \text{Theorem 1.} \]

If $(H_1, TP_1) : \tau$, then either $(H_1, TP_1)$ is a terminal global configuration or $(H_1, TP_1) \rightarrow (H_2, TP_2)$ for some global configuration $(H_2, TP_2)$.

\[ \text{Proof.} \]

We show that an MTMJ transition exists for each well-typed nonterminal global configuration. The proof is by cases of the closed frame of the current thread $th$.

- $CF_{th} = th’.\texttt{start}()$:

  Since $TP_1$ is well-typed, the local configuration of $th$ is also well-typed. Therefore, the rule TE-Method derives a type for $th’.\texttt{start}()$. In this rule, the type of $th’$, its type in the heap, is supposed to be $C$ which is obtained using the function context. Moreover, the premises $C \leftarrow \texttt{Thread}$ and $th’ \notin \text{dom}(TP_1)$ are satisfied—note that $\Phi_{th}$ is a superset of $\text{dom}(TP_1)$, as implied by the rule TG-TP. Therefore, a reduction is possible using the rule E-MethodVoidStart.

- $CF_{th} \neq th’.\texttt{start}()$:

  In this case, we show that there is a reduction through the rule E-Translate. First, it should be shown that a transition $\rightarrow$ can take place. To do so, we should prove that the local configuration of $th$ is well-typed, i.e., $\Delta \vdash (H_1, VS_{th}^{th}, CF_{th}^{th}, FS_{th}^{th}) : \tau$, so that an MJ transition can occur. Thus, we should establish the following judgments.

  \[
  \begin{align*}
  & - \Delta \vdash H_1 \text{ ok.} \\
  & - \Delta, H_1 \vdash VS_{th}^{th} \text{ ok.} \\
  & - \Delta, H_1, VS_{th}^{th} \vdash CF_{th}^{th} \circ FS_{th}^{th} : \texttt{void} \rightarrow \tau.
  \end{align*}
  \]

  The first and second judgments are directly derived from TG-CFG and TG-TP by assuming $(H_1, TP_1) : \tau$. Moreover, according to TG-TP, we have $\Delta, H_1, VS_{th}^{th} \vdash CF_{th}^{th} \circ FS_{th}^{th} : \texttt{void} \rightarrow \tau$ by assuming $\Delta, H_1 \vdash TP_{th} : \tau$. The third judgment above is derivable because considering the set $\Phi_{th}$ in the typing context enforces some new restrictions and does not remove any restriction enforced by the typing rules of MJ. Therefore, we can conclude that $\Delta, H_1, VS_{th}^{th} \vdash CF_{th}^{th} \circ FS_{th}^{th} : \texttt{void} \rightarrow \tau$. It should be noted that $CF_{th}$ or $FS_{th}$ may be modified by the function $\text{subs}$, as seen in Fig. 7. Similarly, we have $\Delta, H_1, VS_{th}^{th} \vdash CF_{th}^{th} \circ FS_{th}^{th} : \texttt{void} \rightarrow \tau$ by assuming $\Delta, H_1, VS_{th}^{th} \vdash CF_{th}^{th} \circ FS_{th}^{th} : \texttt{void} \rightarrow \tau$ such that $CF_{th}$ and $FS_{th}$ are obtained from the function $\text{subs}$ when it modifies $CF_{th}$ and $FS_{th}$. To arrive at $\Delta, H_1, VS_{th}^{th} \vdash CF_{th}^{th} \circ FS_{th}^{th} : \texttt{void} \rightarrow \tau$, it is shown that a frame itself, closed or open, is typable if it is modified by $\text{subs}$ and the modified frame is typable. Note that a frame stack is typable if its frames are typable. For $x = \bullet$, as seen in Fig. 8, $\text{subs}$ replaces $\bullet$ by either new $C(*)$ when this is the frame preceding $x = \bullet$ or “new $C(v_1,\ldots,v_n)$” when the
The proof is by case analysis on MTMJ transition relation $\rightarrow$.

1. $\Delta \vdash H_1$ ok.
2. $\Delta, H_1 \vdash VS_{th}^{1}$ ok.
3. $\Delta, H_1, VS_{th}^{1}, \Phi_{th}^{1} \vdash CF_{th}^{1} \circ FS_{th}^{1} : \text{void} \rightarrow \tau$.

To show that $(H_2, TP_2)$ is typable, we should prove the following judgments.

4. $\Delta \vdash H_2$ ok.
5. $\Delta, H_2 \vdash VS_{th}^{2}$ ok.
6. $\Delta, H_2, VS_{th}^{2}, \Phi_{th}^{2} \vdash CF_{th}^{2} \circ FS_{th}^{2} : \text{void} \rightarrow \tau'$.

From (3), it is concluded that $\Delta, H_1, VS_{th}^{1} \vdash CF_{th}^{1} \circ FS_{th}^{1} : \text{void} \rightarrow \tau$ even if $CF$, or $FS$, is modified by the function $subs$. Thus, (1) and (2), imply
that \((H_1, VS_{th}, CF_{th}, FS_{th}) : \tau\), a well-typed configuration of MJ. As MJ preserves types, (4) and (5) hold for the current thread \(th\). Moreover, we have \(\Delta, H_2, VS_{th}^2 \vdash CF_{th}^2 \circ FS_{th}^2 : \text{void} \rightarrow \tau' \) with \(\tau' \prec \tau\). For (6), we investigate the transitions of MJ to show that adding \(\Phi_{2,th}\) to the context for type checking \(CF_{th}^2 \circ FS_{th}^2\) preserves the type of the composition. Since \(\text{dom}(TP)\) does not change, \(\Phi_{2,th}\) may differ from \(\Phi_{1,th}\) if such a transition results in modifying the heap or the variable stack. In what follows, we prove that \(CF_{th}^2 \circ FS_{th}^2\) is typable even in the presence of restrictions imposed by \(\Phi_{2,th}\). As MJ preserves types, \(\text{void} \rightarrow \tau'\) would then be the type of this composition.

- (E-Skip): Assuming \(\Delta, H, VS_{th}, \Phi_{1,th}^1 \vdash (\_ \circ F \circ FS) : \text{void} \rightarrow \tau\) for the case where \(CF_{th}^1, \text{is "}\_\text{" and } FS_{th}^1 = F \circ FS\), we should prove that \(\Delta, H, VS_{th}, \Phi_{1,th}^2 \circ F \circ FS) : \text{void} \rightarrow \tau\). According to TF-STACKCLOSED and TS-NoOP, the judgments \(\Delta; \text{context}(H, VS_{th}); \Phi_{1,th}^1 \vdash \text{void} \circ FS\) and \(\Delta, H, VS_{th}, \Phi_{1,th}^1 \circ \text{void} : FS : F \circ FS : \text{void} \rightarrow \tau\) hold. Note that neither "_" nor \(F \circ FS\) are modified through \(\text{subs}\) because \(F\) is not an open frame, for the type of \(\_\) cannot be set to \(\text{void}\). To get the result, therefore, it is enough to show that \(\Phi_{1,th}^2 = \Phi_{2,th}^2\). This equality is satisfied, because \(\text{dom}(TP)\), \(H\), and \(VS_{th}\) are not changed by E-Skip. In this way, what is returned by \(\text{giveThrads}\) does not change as well.

- (E-Sub) with \(F = CF\): We assume \(\Delta, H, VS_{th}, \Phi_{1,th}^1 \vdash (v \circ CF \circ FS) : \text{void} \rightarrow \tau\), where \(CF_{th}^1 = v\) and \(FS_{th}^1 = CF \circ FS\), and prove that \(\Delta, H, VS_{th}, \Phi_{1,th}^2 \circ \text{CF} \circ \text{FS} : \text{void} \rightarrow \tau\). Since \(CF\) does not contain \(\_\), the substitution of \(v\) for \(\_\) results in \(CF\) itself. The value \(v\) is either \text{null} or an object identifier \(o\). According to TF-STACKCLOSED and TE-NULL or TE-OID, we have \(\Delta; \text{context}(H, VS_{th}); \Phi_{1,th}^1 \vdash v : \tau'' \circ \Phi_{1,th}^1\) and \(\Delta, H, VS_{th}, \Phi_{1,th}^1 \circ \text{CF} \circ \text{FS} : \tau'' \rightarrow \tau\). Note that the frame stack \(FS_{th}^1\) is not modified by the function \(\text{subs}\). This is because this function modifies only those frame stacks that begin with an open frame. Therefore, it will suffice to show that \(\Phi_{1,th}^2 = \Phi_{2,th}^2\). This holds because of equal \(H, VS_{th}\), and \(\text{dom}(TP)\) before and after the application of E-Sub with \(F = CF\).

- (E-Sub) with \(F = OF\): By assuming \(\Delta, H, VS_{th}, \Phi_{1,th}^1 \vdash (v \circ OF \circ FS) : \text{void} \rightarrow \tau\), where \(CF_{th}^1 = v\) and \(FS_{th}^1 = OF \circ FS\), we prove that \(\Delta, H, VS_{th}, \Phi_{1,th}^2 \circ FS \vdash \text{OF} \circ \text{FS} : \text{void} \rightarrow \tau\). According to TF-STACKCLOSED, we have \(\Delta; \text{context}(H, VS_{th}); \Phi_{1,th}^1 \vdash v : \tau'' \circ \Phi_{1,th}^1\) and \(\Delta, H, VS_{th}, \Phi_{1,th}^1 \vdash \text{OF} \circ \text{FS} : \tau'' \rightarrow \tau\), where \(OF \circ FS\) may be modified by the function \(\text{subs}\). Note that \(\Phi_{1,th}^2 = \Phi_{2,th}^2\) because \(H, VS_{th}\), and \(\text{dom}(TP)\) remain unchanged before and after the transition.

* If \(OF\) is modified by \(\text{subs}\) and it is not “\(x = \_\)" or “\(v'.f = \_\)" and does not begin with \(\_g\), it can be easily shown that the result is \([v/\_]\circ \text{OF}\). It should be noted that \(\text{FS}\) is not modified by \(\text{subs}\) in this case. For instance, if \(OF = \text{new } C(v_1, \ldots, v_n, \_e_{i+1}, \ldots, e_n)\) and \(FS = (x = \_ \circ FS)\), \(FS\) is not modified by \(\text{subs}\) because \(\Delta_{th}(C)\) is not true. If \(\Delta_{th}(C)\) is true, the body of the constructor of \(C\) should contain no invocation of \text{start}, while such an invocation has resulted in the modification of \(OF\) by \(\text{subs}\) (see the rules TE-NEW and T-C-DEFN.)

* If \(OF \circ FS\) is changed by \(\text{subs}\) and it begins with \(\_g\), i.e., it is of the form \((\_g, \_g, \ldots, \_g) \circ OF' \circ FS'\), then the result of the application of
subs is \((\bullet g_1 \circ \ldots \circ \bullet g_n) \circ [v.g_1.g_2.\ldots.g_n/\bullet]OF' \circ FS'\) which is typable against \(\Phi^*_{th}\). The term \(v.g_1 \circ (\bullet g_2 \circ \ldots \circ \bullet g_n) \circ [v.g_1.g_2.\ldots.g_n/\bullet]OF' \circ FS'\) is also typable against \(\Phi^*_{th}\), because replacing \(\bullet\) with \(v\) in \(\bullet g_1\) preserves the type. Note that the substitution of \(v.g_1.g_2.\ldots.g_n\) for \(\bullet\) is performed by \(subs\) when both \((v)\circ OF \circ FS\) and \([v/\bullet]OF \circ FS\) are type checked. As with the previous case, \(FS'\) does not change when \(subs\) is applied.

* If \(v = o\) and \(OF\) is \("x = \bullet\) or \("v'.f = \bullet\)”, then the result of \([v/\bullet]OF\) is \("x = o\) or \("v'.f = o\)”. Note that the judgment \(\Delta, H, VS_{th}, \Phi^*_{th}, [v/\bullet]OF \circ FS : \text{void} \rightarrow \tau\) is type checked by TF-StackClosed, as the result of \([v/\bullet]OF\) is a closed frame. When \((v)\circ OF \circ FS\) is type checked, if the \(\bullet\) of \(OF\) is replaced by \(\text{new} C(v_1, \ldots, v_n)\) through \(subs\), then \(v\) in \([v/\bullet]OF\) is also replaced by the same expression when \([v/\bullet]OF \circ FS\) is type checked. Therefore, the same frame is type checked in both cases. It is worth noting that \(FS\) is not changed here because the type of \(OF\) is \(\text{void}\) so that the first frame of \(FS\) cannot be an open frame.

* If \(OF\) is not changed by \(subs\), \(\Delta, H, VS_{th}, \Phi^*_{th}, OF \circ FS : \text{void} \rightarrow \tau\) is type checked by the rule TF-StackOpen. In this way, we have \(\Delta : \text{context}(H, VS_{th}), \bullet : \tau'^\prime, \Phi^*_{th}, OF : \tau_1|\Phi'\). Therefore, it can be concluded that \(\Delta : \text{context}(H, VS_{th}), \Phi^*_{th}, [v/\bullet]OF : \tau_1|\Phi'\). Moreover, the set \(\Phi'\) remains unchanged because neither \(\bullet\) nor the value \(v\) change this set. Finally, we show that \(\Delta, H, VS_{th}, \Phi' \circ FS : \tau_1 \rightarrow \tau\). If the first frame of \(FS\) is not modified by \(subs\) when \(OF\) or \([v/\bullet]OF\) is the frame preceding \(FS\), the result is immediate. Otherwise, for the first frame \(F'\) which is either \("x = \bullet\) or \("v'.f = \bullet\)”, and \(OF = \text{new} C(*)\), it should be shown that \(\Delta, H, VS_{th}, \Phi' \circ [OF/\bullet]F' \circ FS' : \tau_1 \rightarrow \tau\) holds if \(\Delta, H, VS_{th}, \Phi' \circ [OF/\bullet]F' \circ FS' : \tau_1 \rightarrow \tau\). It can be easily shown that this is derivable. In particular, \(\text{new} C(*)\) does not contain any invocation of \(\text{start}\) because \(\Delta_{th}(C)\) is true, a condition required for the modification of \(F'\).

- (E-Return) By assuming \(\Delta, H, MS \circ VS, \Phi^*_{th}, \text{return }v_1) \circ FS : \text{void} \rightarrow \tau\), where \(VS^2_{th} = MS \circ VS\), we prove that \(\Delta, H, VS, \Phi^*_{th}, \text{return }v_1) \circ FS : \text{void} \rightarrow \tau\). According to TF-StackMethod2, we have \((7)\Delta : \text{context}(H, MS \circ VS), \Phi^*_{th}, \text{return }v_1) \circ FS : \text{void} \rightarrow \tau\). According to TF-StackClosed, thus, we should prove that \((9)\Delta : \text{context}(H, VS), \Phi^*_{th}, \text{return }v_1) \circ FS : \text{void} \rightarrow \tau\). According to TF-StackOpen, \(FS = \bullet \circ FS'\) or \(FS = (v'.f = \bullet) \circ FS'\): If \(v\) satisfies the conditions for the application of \(subs\), the first \(\bullet\) of \(FS\) is replaced by \(\text{new} C(v_1, \ldots, v_n)\) in both \((8)\) and \((10)\). Since \(\text{giveThreads}\) may not return some expressions that are in \(\Phi^*_{th}\), we have \(\Phi^*_{th} \subseteq \Phi^*_{th}\). Since \(v\) and \(FS\) are typable according to \((7)\) and \((8)\), by assuming a larger set of restrictions \(\Phi^*_{th}\), \((9)\) and \((10)\) would be derivable.

- Other forms of \(FS\) modified by \(subs\), e.g., \(FS = \bullet \circ FS'\). In these cases, it can be easily shown that the frame stack resulting from modification is the same in both \((8)\) and \((10)\). For example, if \(FS = (\bullet g_j \circ)_{j=1}^k \bullet \circ FS'\), then the resulting \(FS\) is \((\bullet g_j \circ)_{j=1}^k \circ FS'\) when the frame preceding \(FS'\)
is either "return v;" or "v". As with the previous case, we have \( \Phi_{th'}^0 \subseteq \Phi_{th'}^1 \). Therefore, (9) and (10) are derivable.

* For other forms of FS not changed by subs, we also have \( \Phi_{th'}^2 \subseteq \Phi_{th'}^1 \).

Therefore, (9) and (10) are derivable under assumptions (7) and (8).

- (E-VarAccess) By assuming \( \Delta, H, MS \circ VS, \Phi_{th'}^1 \vdash (x) \circ FS : \text{void} \rightarrow \tau \) with \( VS_{th'}^1 = MS \circ VS \), we prove that \( \Delta, H, MS \circ VS, \Phi_{th'}^2 \vdash (v) \circ FS : \text{void} \rightarrow \tau' \), where \( \tau' < \tau \). According to TF-StackClosed, we have (11) \( \Delta; \text{context}(H, MS \circ VS); \Phi_{th'}^0 \vdash x : \tau_1 \Phi_{th'}^1 \) and (12) \( \Delta, H, MS \circ VS, \Phi_{th'}^1 \vdash FS : \tau_1 \rightarrow \tau \). To get the result, it should be shown that (13) \( \Delta; \text{context}(H, MS \circ VS); \Phi_{th'}^0 \vdash v : \tau_2 \Phi_{th'}^2 \) and (14) \( \Delta, H, MS \circ VS, \Phi_{th'}^2 \vdash FS : \tau_2 \rightarrow \tau' \). Since \( MS \circ VS \) and \( \text{dom}(TP) \) remain unchanged before and after this reduction, we have \( \Phi_{th'}^2 = \Phi_{th'}^1 \).

Note that (13) is derived trivially. Moreover, (14) is directly derivable from (12) if \( FS \) is such that no modification by \( subs \) is required. The following argument is about those frame stacks FS that are modified by subs.

* If FS is either \( (y = \_ \_ \_ \_ ) \circ FS' \) or \( (v'.f = \_ \_ \_ \_ ) \circ FS' \), it is not modified by subs. This is because \( \text{eval}(MS, x) = v \), and thus the conditions for applying \( subs \) are not satisfied.

* If FS is \( (\_ \_ \_ \_ \_ \_ g_j)_{j=1}^k \circ \text{start}() \circ FS' \), then \( subs \) replaces it by \( (\_ \_ \_ \_ \_ \_ g_j)_{j=1}^k x \_ \_ \_ \_ \_ \_ g_j_{j=1}^k \text{start}() \circ FS' \) in (12) and by \( (\_ \_ \_ \_ \_ \_ g_j)_{j=1}^k v.(g_j)_{j=1}^k \_ \_ \_ \_ \_ \_ g_j_{j=1}^k \text{start}() \circ FS' \) in (14). The judgment (12) implies \( x.(g_j)_{j=1}^k \notin \Phi_{th'}^1 \), which, in turn, results in \( v.(g_j)_{j=1}^k \notin \Phi_{th'}^2 \). Another result of (12) is that the type of \( x.(g_j)_{j=1}^k \) is a subclass of \( \text{Thread} \). From (5), for the current thread \( th' \), the type of \( v.(g_j)_{j=1}^k \) is also a subclass of the type of \( x.(g_j)_{j=1}^k \), and therefore, it is a subclass of \( \text{Thread} \) as well. It can be easily shown that the other premises of TE-METHOD are also satisfied for \( v.(g_j)_{j=1}^k \_ \_ \_ \_ \_ \_ g_j_{j=1}^k \text{start}() \). Therefore, (14) is derivable.

* We now consider other forms of FS that are modified by subs. For \( \text{new} \ C(v_1, \ldots, v_{i-1}, \_ \_ \_ \_ \_ \_ e_{i+1}, \ldots, e_n) \circ FS' \) for example, the resulting frame stack is \( \text{new} \ C(v_1, \ldots, v_{i-1}, x, e_{i+1}, \ldots, e_n) \circ FS' \) in (12) and \( \text{new} \ C(v_1, \ldots, v_{i-1}, v, e_{i+1}, \ldots, e_n) \circ FS' \) in (14). According to TE-New, \( x \notin \Phi_n \) because (12) is derivable. Since \( \Phi_{th'}^1 \subseteq \Phi_n \), we have \( x \notin \Phi_{th'}^1 \) as well. Therefore, its value \( v \) is not in \( \text{dom}(TP) \) \( \subseteq \Phi_{th'}^1 \).

Since type checking \( v_1, \ldots, v_{i-1} \) does not modify \( \Phi_{th'}^1 \) and \( e_{i+1}, \ldots, e_n \) do not contain any value, we have \( v \notin \Phi_n \). Therefore, (14) can be derived. Other forms are treated similarly. It is worth noting that if \( FS = \_ \_ \_ \_ m(e) \circ FS' \) and this \( \in \Phi_m \), then \( m \) is not \_ \_ \_ \_ start. This is because (12) cannot be derivable otherwise.

Note that if \( FS = ((C) \_ \_ \_ \_ FS') \), it is supposed that \( \neg \Delta_{th}(\tau_1) \). To derive (14), we should have \( \neg \Delta_{th}(\tau_2) \) that is ensured according to the definition of \( \Delta_{th} \) and \( \tau_2 < \tau_1 \).

- (E-VarWrite) By assuming \( \Delta, H, MS \circ VS, \Phi_{th'}^1 \vdash (x = v_1) \circ FS : \text{void} \rightarrow \tau \) with \( VS_{th'}^1 = MS \circ VS \), we prove that \( \Delta, H, MS' \circ VS, \Phi_{th'}^2 \vdash (v) \circ FS : \text{void} \rightarrow \tau \) in which \( MS' = update(MS, x \mapsto v) \)—this updates \( MS \) by \( v \) as the value of \( x \). According to TF-StackClosed, we have (15) \( \Delta; \text{context}(H, MS \circ VS); \Phi_{th'}^0 \vdash x = v_1 : \text{void}; \Phi_{th'} \) and (16) \( \Delta, H, MS \circ VS, \Phi_{th'}^1 \vdash FS : \text{void} \rightarrow \tau \). Now, we should prove (17) \( \Delta; \text{context}(H, MS' \circ VS); \Phi_{th'}^0 \vdash : \text{void}; \Phi_{th'}^2 \) and (18) \( \Delta, H, MS' \circ VS, \Phi_{th'}^2 \vdash FS : \text{void} \rightarrow \tau \).
The judgment (17) is derived trivially. Since the type of $\bullet$ cannot be $\text{void}$, $FS$ does not begin with an open frame, and therefore, it is not modified by $\text{subs}$. We show $\Phi_{th}^{1} \subseteq \Phi_{th}^{2}$, and therefore, (18) is derivable from (16). $\Phi_{th}^{2}$ consists of $\text{dom}(TP)$—not changing before and after $\text{E-VarIntro}$—and the result of the function $\text{giveThreads}$. It should be shown that what is returned by this function when the method scope is $MS'$ does not contain an expression that is not returned when the method scope is $MS$. Since the only change to $MS$ is assigning $v$ to $x$, this value—as an object identifier—or the values of its fields may be in $\text{dom}(TP)$ so that $x\orb f_{1}\orb f_{2}\gap f_{n}$ is returned as a new expression. If so, $\Delta_{th}(C')$ is true, where $C'$ is the type of $v$. If $v \neq \text{null}$, $v$ is the value of an expression of the form $\text{new} C'(\bar{v})$. This is because of (15) asserting $x = v$; is typable. Such an expression, as stated by Lemma 4, returns an object identifier which neither itself nor its fields are in $\text{dom}(TP)$. Therefore, neither $v$ nor its fields may be in $\text{dom}(TP)$.

- (E-VarIntro) We assume $\Delta,H,(BS\circ MS)\circ VS,\Phi_{th}^{2} \vdash (C;\, x) \circ FS : \text{void} \rightarrow \tau$ with $VS_{th}^{1} = (BS \circ MS) \circ VS$ and prove that $\Delta,H,(BS\circ MS)\circ VS,\Phi_{th}^{2} \vdash (\; \circ FS : \text{void} \rightarrow \tau$, where $BS' = BS[x \mapsto (\text{null}, C)]$. According to the extended version of the rule $\text{TF-StackIntro}$ borrowed from MIJ, we have (19) $\Delta,H,(BS' \circ MS)\circ VS,\Phi_{th}^{1} \vdash FS : \text{void} \rightarrow \tau$. From $\text{TF-StackClosed}$, it should be shown that (20) $\Delta;\text{context}(H,(BS' \circ MS)\circ VS);\Phi_{th}^{2} \vdash ;\text{void};\Phi$ and (21) $\Delta,H,(BS'\circ MS)\circ VS,\Phi_{th}^{1} \vdash :FS : \text{void} \rightarrow \tau$. The judgment (20) is derived trivially. Note that $\text{giveThreads}$ does not return $x$ as a new expression. This is because its value is $\text{null}$ which may not be in $\text{dom}(TP)$. Since the set returned by $\text{giveThreads}$ remains unchanged, we have $\Phi_{th}^{1} = \Phi_{th}^{2}$. Therefore, (21) can be derived from (19).

- (E-IfI) By assuming $\Delta,H,VS,\Phi_{th}^{2} \vdash (\text{if} \, (v_{1} = v_{2}) \{ \bar{s}_{1} \} \text{ else } \{ \bar{s}_{2} \}) \circ FS : \text{void} \rightarrow \tau$, it should be proved that $\Delta,H,VS,\Phi_{th}^{2} \vdash \{ \bar{s}_{1} \} \circ FS : \text{void} \rightarrow \tau$. According to $\text{TF-StackClosed}$, we have (22) $\Delta;\text{context}(H,VS);\Phi_{th}^{2} \vdash (\text{if} \, (v_{1} = v_{2}) \{ \bar{s}_{1} \} \text{ else } \{ \bar{s}_{2} \}) \circ \text{void}\Phi'$ and (23) $\Delta,H,VS,\Phi' \vdash FS : \text{void} \rightarrow \tau$. To get the result, we should prove (24) $\Delta;\text{context}(H,VS);\Phi_{th}^{1} \vdash \{ \bar{s}_{1} \} ;\text{void};\Phi''$ and (25) $\Delta,H,VS,\Phi'' \vdash FS : \text{void} \rightarrow \tau$. Since $H$, $VS$, and $\text{dom}(TP)$ remain unchanged by E-IfI, we have $\Phi_{th}^{1} = \Phi_{th}^{2}$. According to $\text{TS-IF}$ and (22), we have $\Delta;\text{context}(H,VS);\Phi_{th}^{1} \vdash \{ \bar{s}_{1} \} ;\text{void};\Phi'$. Note that since both of the expressions in the condition part of if are values, $\Phi_{th}^{1}$ does not change as the result of their type checking which takes place prior to type checking $\{ \bar{s}_{1} \}$. Thus, (24) is derived. Moreover, from $\text{TS-IF}$, $\Phi'' \subseteq \Phi'$. Since $FS$ is typable against $\Phi'$, it is also typable against $\Phi''$ which results in (25).

- (E-If2) Similar to E-IfI.

- (E-BlockIntro) By assuming $\Delta,H,MS \circ VS,\Phi_{th}^{2} \vdash \{ \bar{s} \} \circ FS : \text{void} \rightarrow \tau$ with $VS_{th}^{1} = MS \circ VS$, we prove that $\Delta,H,(\{ \} \circ MS) \circ VS,\Phi_{th}^{2} \vdash (\{ \} \circ FS : \text{void} \rightarrow \tau$, where $VS_{th}^{1} = (\{ \} \circ MS) \circ VS$. According to $\text{TF-StackClosed}$, we have (26) $\Delta;\text{context}(H,MS \circ VS);\Phi_{th}^{1} \vdash (\{ \} \circ \text{void}\Phi' \circ FS : \text{void} \rightarrow \tau$. According to $\text{TS-Block}$ and from (26), we have $\Delta;\text{context}(H,MS \circ VS);\Phi_{th}^{1} \vdash \{ \} ;\text{void};\Phi'$. Since only an empty block is pushed to $MS$ in $VS_{th}^{2}$, the result of $\text{giveThreads}$ remains the same, compared to $VS_{th}^{1}$. Therefore, we have $\Phi_{th}^{1} = \Phi_{th}^{2}$. For the same reason, we can deduce (28) $\Delta;\text{context}(H,(\{ \} \circ MS) \circ VS);\Phi_{th}^{1} \vdash \{ \} ;\text{void};\Phi'$. 
A lemma has been proved in MJ which states that having $\Delta, H, MS \circ VS \vdash FS : \text{void} \rightarrow \tau$ and $\Delta; \text{context}(H, (BS \circ MS) \circ VS) \vdash \bar{s} : \text{void}$, the judgment $\Delta, H, (BS \circ MS) \circ VS \vdash \bar{s} \circ \{\} \circ FS : \text{void} \rightarrow \tau$ can be derived. It can be easily proved that this claim is also fulfilled for the typing judgments of MTMJ. In other words, if we have $\Delta, H, MS \circ VS, \Phi_2 \vdash FS : \text{void} \rightarrow \tau$ and $\Delta; \text{context}(H, (BS \circ MS) \circ VS), \Phi_1 \vdash \bar{s} : \text{void} \Phi_2$, then we can also deduce $\Delta, H, (BS \circ MS) \circ VS, \Phi_1 \circ \bar{s} \circ \{\} \circ FS : \text{void} \rightarrow \tau$. In this way, from (27) and (28), (29) $\Delta, H, \{\} \circ MS \circ VS, \Phi_{1h} \vdash \bar{s} \circ \{\} \circ FS : \text{void} \rightarrow \tau$. This completes the proof in the case.

- (E-Cast) The frame stack $H$ derivable. Since the type of $\text{context}$, $H \circ MS \circ VS \circ \Phi_2$ and $\Phi_2 \circ \bar{s} \circ \{\} \circ FS : \text{void} \rightarrow \tau$, where $\tau' < \tau$. According to TF-StackClosed, we have (30) $\Delta; \text{context}(H, VS); \Phi_{1h} \vdash (C_1) o : C_1 \Phi_{1h}$ and (31) $\Delta, H, VS, \Phi_{1h} \circ FS : C_1 \rightarrow \tau$. Now, we should show that (32) $\Delta; \text{context}(H, VS); \Phi_{1h} \vdash o : C_2 \Phi_{1h}$ and (33) $\Delta, H, VS, \Phi_{1h} \circ FS : C_2 \rightarrow \tau'$. We have $\Phi_{1h} = \Phi_2$ as the result of equal $H, VS$, and $\text{dom}(TP)$ before and after applying E-Cast. The frame stack $FS$ in (31) is not modified by $\text{subs}$ because $(C_1) o$ or $(C_1) o, g_1, g_2, \ldots, g_n$ are not of type $\text{Thread}$, as stated in Lemma 1. Moreover, the assignment of $(C_1) o$ or $(C_1) o, g_1, g_2, \ldots, g_n$ when $\Delta h$ is true for the type of these expressions is not typable through the rules $\text{TS-VarWrite}$ and $\text{TS-FieldWrite}$. According to TF-UpCast and from (30), we have $\Delta; \text{context}(H, VS); \Phi_{1h} \circ o : C_2 \Phi_{1h}$. Therefore, (32) is derivable. Note that form (30) we also have $\Delta; \text{context}(H, VS); \Phi_{1h} \circ o : C_2 \Phi_{1h}$. Therefore, the frame stack $FS$ in (33) is not modified by $\text{subs}$ and (33) can be derived from (31).

- (E-FieldWrite) We prove that $\Delta, H, VS, \Phi_{1h} \vdash (o.f) v : FS : \text{void} \rightarrow \tau$ implies $\Delta', H', VS, \Phi_{1h'} \vdash (\cdot) \circ FS : \text{void} \rightarrow \tau'$, where $\tau' < \tau$. According to TF-StackClosed, we have (34) $\Delta; \text{context}(H', VS); \Phi_{1h'} \vdash o.f = v : \text{void} \Phi_{1h'}$ and (35) $\Delta, H', VS, \Phi_{1h'} \circ FS : \text{void} \rightarrow \tau$. Now, we should prove that the following judgments are derivable. (36) $\Delta; \text{context}(H', VS); \Phi_{1h'} \vdash \text{void} \Phi_{1h'}$, and (37) $\Delta, H', VS, \Phi_{1h'} \circ FS : \text{void} \rightarrow \tau$. The judgment (36) is trivially derivable. Since the type of $\bullet$ cannot be set to $\text{void}$, $FS$ may not begin with an open frame, it is not modified by $\text{subs}$. We show that $\Phi_{1h'} \subseteq \Phi_{1h}$, which itself results in (37) by assuming (35). It should be shown that for the argument $H'$, give $\text{Threads}$ does not return a new expression which is not returned when the argument is $H$. Since the only change to $H$ is the assignment of $v$ to $o.f$ and this value or its fields may be in $\text{dom}(TP)$, the extra expressions $x.f$ or $x.f.f_1.f_2.\ldots.f_n$ for $x$ with the value $o$ may be returned by $\text{give} \text{Threads}$. In this way, $\Delta_{1h}(C')$ is true, where $C'$ is the type of $v$. If $v \neq \text{null}$, $v$ is the value of an expression of the form new $C'(\bar{s})$. This is because in (34) we assume that $o.f = v$ is typable. Such an expression form, as stated by Lemma 4, returns an object identifier, e.g., $v$, which itself and its fields are not in $\text{dom}(TP)$.

- (E-FieldAccess) We prove that $\Delta, H, VS, \Phi_{1h} \vdash (o.f) v : FS : \text{void} \rightarrow \tau$, where $H(o) = (C, F)$ and $F(f) = v$, implies $\Delta, H, VS, \Phi_{1h} \vdash (v) \circ FS : \text{void} \rightarrow \tau'$, where $\tau' < \tau$. According to TF-StackClosed, we have (38) $\Delta; \text{context}(H, VS); \Phi_{1h} \vdash o.f : C_2 \Phi_{1h}$ and (39) $\Delta, H, VS, \Phi_{1h} \circ FS : C_2 \rightarrow \tau$. Now, it should be shown that the following judgments are derivable.
is because there exists an object identifier \( o \) having a field \( f \) with the value \( v \). For other forms of \( FS \) that may be modified by \( sub \), we can derive (41) from (39). The justification is similar to that of the case of E-VarAccess.

- (E-New) We prove that \( \Delta; \text{context}(H, VS); \Phi^{1}_{th}; \vdash (\text{new} \ C(v_{1}, \ldots, v_{n})) \circ \Phi^{2}_{th}; \text{void} \rightarrow \tau \)

implies \( \Delta'; \text{context}(H', MS \circ VS); \Phi^{1}_{th}; \vdash [\text{this}_C/\text{this}](\text{super}(\bar{e}); \bar{s}) \circ (\text{return } \alpha) \circ \Phi; \text{void} \rightarrow \tau \), where \( H' = H[\alpha \mapsto (C, \bar{e})], \alpha \notin \text{dom}(H), \bar{v} = \{ f \mapsto \text{null} | f \in \text{dom}(\Delta(C)) \}, MS = (\text{this}_C \mapsto (o, C), x_{1} \mapsto (v_{1}, C_{1}), \ldots, x_{n} \mapsto (v_{n}, C_{n})) \circ [\],\text{cbody}(C) = (x_{1}, \ldots, x_{n}, \text{super}(\bar{e}); \bar{s}) \), and \( \Delta_{C} = C_{1}, \ldots, C_{n}, \Phi_{C} \). Here, as with the type system of MTMJ, we use \( \alpha \)-renaming to distinguish between different occurrences of \( \text{this} \). According to \( \text{TF-StackClosed} \), we have \( (42) \Delta; \text{context}(H, VS); \Phi^{1}_{th}; \vdash \text{new} \ C(v_{1}, \ldots, v_{n}) : C \rangle \Phi' \)

and \( (43) \Delta; H, VS, \Phi' \vdash FS : C \rightarrow \tau \). It has been proved in MJ that the type of a frame stack is preserved by extending heap. That is, by knowing \( \Delta; H, VS \vdash FS : \tau' \rightarrow \tau \) and \( \Delta \vdash C \ ok \), it can be concluded that \( \Delta; H[\alpha \mapsto (C, \bar{e})], VS \vdash FS : \tau' \rightarrow \tau \) for an object identifier \( o \notin \text{dom}(H) \). This can be easily justified in the presence of \( \Phi \) as part of typing context. In other words, if we have \( \Delta; H, VS, \Phi \vdash FS : \tau' \rightarrow \tau \) and \( \Delta \vdash C \ ok \), then we also have \( \Delta; H[\alpha \mapsto (C, \bar{e})], VS, \Phi \vdash FS : \tau' \rightarrow \tau \) for an object identifier \( o \notin \text{dom}(H) \).

Therefore, from (43), we have \( (44) \Delta; H', VS, \Phi' \vdash FS : C \rightarrow \tau \). According to the rule \( T-CDefn \), the body of constructor is typable against \( \Phi = \emptyset \). That is, we have \( \Delta; \Gamma; \emptyset \vdash [\text{this}_C/\text{this}](\text{super}(\bar{e}); \bar{s}) : \text{void} \rightarrow \Phi_{C} \), where \( \Gamma = x_{1} : C_{1}, \ldots, x_{n} : C_{n}, \text{this}_C : C \). In this way, we also have \( \Delta; \text{context}(H', MS \circ VS); \emptyset \vdash [\text{this}_C/\text{this}](\text{super}(\bar{e}); \bar{s}) : \text{void} \rightarrow \Phi_{C} \) considering the definition of context. In effect, \( \text{context}(H', MS \circ VS) \) is a superset of \( \Gamma \) where the types of object identifiers in \( H' \) and variables in \( VS \) other than \( \text{this}_C \) and \( x_{1}, \ldots, x_{n} \) are added to \( \Gamma \). We show that the body of constructor remains well-typed when the frame replacement is \( \Phi^{2}_{th}. \) That is, there is no expression in \( \Phi^{2}_{th} \) on which \( \text{start} \) is invoked in the body of constructor.

Note that every invocation of \( \text{start} \) in the body of constructor is assumed to be gathered in \( \Phi_{C} \). According to \( \text{TE-New} \) and by assuming (42), it can be concluded that \( \text{check}(\Phi_{C}, \Phi^{1}_{th}) \) is true, where \( \Phi_{C} \) is the result of replacing \( \text{this} \) with \( \text{this}_C \) and each \( x_{i} \) with the corresponding value \( v_{i} \) in the set \( \Phi_{C} \). When the function \( \text{check} \) returns true, it means that there is not any common expression in both \( \Phi_{C} \) and \( \Phi^{1}_{th} \). Now, we prove (45) \( \Delta; \text{context}(H', MS \circ VS); \Phi^{1}_{th}; [\text{this}_C/\text{this}](\text{super}(\bar{e}); \bar{s}) : \text{void} \rightarrow \Phi' \). As seen, \( H' \) and \( VS^{2}_{th} = MS \circ VS \) are obtained by adding some new elements to \( H \) and \( VS_{th} \). Therefore, we have \( \Phi^{1}_{th} \subseteq \Phi^{2}_{th} \). Since the object identifier \( o \) is new—it is not in \( \text{dom}(H) \)—and its field values are \( \text{null} \), neither \( \text{this}_C \) nor its fields are in \( \Phi^{2}_{th} \). However, \( x_{i} \) or \( x_{i}, f_{1}, f_{2}, \ldots, f_{n} \) may be in \( \Phi^{2}_{th} \). If so, \( v_{i} \) or the value of \( v_{i}, f_{1}, f_{2}, \ldots, f_{n} \) is also in \( \text{dom}(TP) \subseteq \Phi^{1}_{th} \). Since any invocation of \( \text{start} \) on \( v_{i} \) or \( v_{i}, f_{1}, f_{2}, \ldots, f_{n} \) in \( \Phi^{2}_{th} \) is checked successfully against \( \Phi^{1}_{th} \), checking such an invocation on \( x_{i} \) or \( x_{i}, f_{1}, f_{2}, \ldots, f_{n} \) against \( \Phi^{1}_{th} \) is also successful. Note that \( \Phi^{2}_{th} \) contains expressions that are composed of \( v_{1}, \ldots, v_{n} \) as well as those expressions that are composed of \( x_{1}, \ldots, x_{n} \). Therefore, (45) can be derived. Moreover, (46)
\( \Delta; \text{context}(H', MS \circ VS); \Phi^2_{th} \vdash [\text{this}_{th} / \text{this}] \text{super}(e); s; \text{return } \alpha; : C | \Phi'' \) is obtained from (45). According to TE-NEW, we have \( \Phi' = \Phi^1_{th} \cup \Phi_C \). Moreover, from T-CDefn, we have \( \Phi'' = \Phi^2_{th} \cup \Phi_C \). Not considering those expressions that are composed of formal parameters \( x_1, \ldots, x_n \), we have \( \Phi'' \subseteq \Phi' \).

In this way, from (44), it can be concluded that (47) \( \Delta, H', VS, \Phi'' \vdash FS : C \rightarrow \tau \). Note that \( FS \) remains well-typed considering those expressions that are composed of \( x_1, \ldots, x_n \). This is because existing erroneous double invocation of \text{start} on such an expression in \( FS \) contradicts with our assumption that the composition of \text{new} \( C(v_1, \ldots, v_n) \) and \( FS \) is typable. Similar to the case of E-BlockIntro, a lemma has been proved in MJ which states that if we have \( \Delta, H, VS \vdash FS : \tau' \rightarrow \tau \) and \( \Delta; \text{context}(H, MS \circ VS); \Phi \vdash \tau' \rightarrow \tau \), then \( \Delta, H, MS \circ VS \vdash s_1 \circ \ldots \circ s_n \circ FS : \text{void} \rightarrow \tau \), where \( s_n \) is of the form “\text{return } e''". This can be extended into MTMJ in a straightforward manner. That is, from \( \Delta, H, VS, \Phi \vdash FS : \tau' \rightarrow \tau \) and \( \Delta; \text{context}(H, MS \circ VS); \Phi_1 \vdash s_1 \circ \ldots \circ s_n \vdash \tau' \), the judgment \( \Delta, H, MS \circ VS, \Phi_1 \vdash s_1 \circ \ldots \circ s_n \circ FS : \text{void} \rightarrow \tau \) with the same form of \( s_n \) is also derivable. In this way, from (46) and (47), it can be concluded that (47) \( \Delta, H', MS \circ VS, \Phi''_{th} \vdash [\text{this}_C / \text{this}] \text{super}(e); s; \text{return } \alpha; : FS \rightarrow \tau \).

- (E-Method) We prove that \( \Delta, H, VS, \Phi^1_{th} \vdash (o.m(v_1, \ldots, v_n)) \circ FS : \text{void} \rightarrow \tau \) implies \( \Delta, H, MS \circ VS, \Phi^2_{th} \vdash [\text{this}_m / \text{this}] (s; \text{return } e); o \circ FS : \text{void} \rightarrow \tau \), where \( H(o) = (C, \Phi_c), C(m) = C_1, \ldots, C_n = \Phi_m \rightarrow C', \text{cnbody}(C, m) = (x_1, \ldots, x_n), s; \text{return } e; \), and \( MS = \{ \text{this}_m \rightarrow (o, C), x_1 \mapsto (v_1, C_1), \ldots, x_n \mapsto (v_n, C_n) \} \cap [\). Similar to the type system of MTMJ, we use \( o \)-renaming to distinguish between different occurrences of \text{this}. According to TF-StackClosed, we have \( \Delta; \text{context}(H, VS); \Phi^1_{th} \vdash (o.m(v_1, \ldots, v_n)) : C | \Phi' \) and \( \Delta, H, VS, \Phi' \vdash : C' \rightarrow \tau \). Now, we should prove that \( \Delta; \text{context}(H, MS \circ VS); \Phi^2_{th} \vdash [\text{this}_m / \text{this}] (s; \text{return } e); C' | \Phi'' \) and \( \Delta, H, VS, \Phi'' \vdash : C' \rightarrow \tau \). The reasoning is similar to the previous case. Note that instead of T-CDefn, the rule T-MDefn is considered here.

- (E-MethodVoid) Similar to the previous case. Note that for \( m = \text{start} \), we should consider the rule E-MethodVoidStart.

- (E-Super) By assuming \( \Delta, H, MS \circ VS, \Phi^1_{th} \vdash (\text{super}(v_1, \ldots, v_n)) \circ FS : \text{void} \rightarrow \tau \), we prove that \( \Delta, H, MS' \circ MS \circ VS, \Phi^2_{th} \vdash (s; \text{return } \alpha); o \circ FS : \text{void} \rightarrow \tau \), where \( MS(\text{this}_C) = (o, C), C \rightarrow C', \Delta_c(C') = C_1, \ldots, C_n, \Phi_C, \text{cnbody}(C') = (x_1, \ldots, x_n, s), \) and \( MS' = \{ \text{this}_C \rightarrow (o, C'), x_1 \mapsto (v_1, C_1), \ldots, x_n \mapsto (v_n, C_n) \} \cap [\). According to TF-StackClosed, we have \( \Delta; \text{context}(H, MS \circ VS); \Phi^2_{th} \vdash \text{super}(v_1, \ldots, v_n) \circ FS : \Phi' \rightarrow \text{void} \) and \( \Delta, H, MS \circ VS, \Phi' \vdash : \text{void} \rightarrow \tau \). To get the result, we should derive \( \Delta; \text{context}(H, MS' \circ MS \circ VS); \Phi^2_{th} \vdash s; \text{return } \alpha; : C | \Phi'' \) and \( \Delta, H, MS \circ VS, \Phi'' \vdash : C' \rightarrow \tau \). The reasoning is similar to the case E-New.

- The transitions of MJ that result in \text{NPE} or \text{CCE}: Type is trivially preserved because these error states can be typable to any type including \( \tau \).

We do not elaborate on the rest of MJ transitions. They are treated similarly. For threads \( th \) other than the current thread, we should show (5) and (6). Note that for such threads, \( VS_{th}, CF_{th} \) and \( FS_{th} \) are not changed.

5. \( \Delta, H_2 \vdash VS_{th}^1 \).
6. \( \Delta, H_2, VS_{th}^1, \Phi^2_{th} \vdash CF_{th}^1 \circ FS_{th}^1 : \text{void} \rightarrow \tau \).
As explained earlier, we have

2. \( \Delta, H_1 \vdash V S_{th} : \text{ok} \) and
3. \( \Delta, H_1, V S_{th} \vdash CF_{th} \circ FS_{th} : \text{void} \rightarrow \tau \).

The heap \( H_2 \) may differ from \( H_1 \) in the following ways:

- \( H_2 = H_1[\alpha' \mapsto (C, \mathbb{F}[f \mapsto v])] \) as the result of “\( \alpha'.f = v' \)”, where \( \alpha' \in \text{dom}(H_1) \).
- \( H_2 = H_1[\alpha' \mapsto (C, \mathbb{F})] \) as the result of “new \( C(v') \)”, where \( \alpha' \notin \text{dom}(H_1) \).

A variable in \( V S_{th} \) may be changed through the reduction of \( CF_{th} \). Since \( CF_{th} \) remains the same when E-Translate is applied, the values of variables in \( V S_{th} \) do not change either. Since the type of a value does not change, (5) is derivable from (2). Moreover, we prove \( \Phi_{th} \subseteq \Phi_{th} \) which itself results in (6) under the assumption (3). Since \( \text{dom}(TP) \) does not change, it is enough to prove that the function \( \text{giveThreads} \) with the argument \( H_2 \) returns a subset of expressions returned when the argument is \( H_1 \). Since \( V S_{th} \) is the same when \( \Phi_{th} \) and \( \Phi_{th} \) are computed, it is obvious that the same subset of local variables is returned by \( \text{giveThreads} \). However, a field \( f \) in \( H_1 \) may be modified through “\( \alpha'.f = v' \)” such that \( v \) is in \( \text{dom}(TP) \), while the old value of \( f \) has not been in this set. If so, the expression on the right side of the assignment is of the form (1) or (2) stated in Lemma 1. In this way, the expression \( \alpha'.f \) in which the value of \( \alpha' \) is \( \alpha' \) and the expression on the right have the same value \( v \) in \( \text{dom}(TP) \). This is in contrast to Corollary 1. To prove (6), moreover, it should be shown that the type \( \tau \) is preserved by \( H_2 \) as part of the typing context. It is obvious that the type of a field remains the same according to TE-FIELD ACCESS even if its value is changed. It can then be shown that the types of other forms remain unchanged. It is worth noting that if \( CF_{th} \) and \( FS_{th} \) are modified by \( \text{subs} \), the result of modification is the same in both (3) and (6).

- (E-MethodVoidStart) For every thread \( th' \) other than \( th \), we have \( \Delta, H \vdash th' : \text{Thread} \) as a result of \( \Delta, H \vdash TP : \tau \). For \( th \), from the premises \( H(th) = (C, \mathbb{F}) \) and \( C \prec \text{Thread} \), it can also be deduced that \( \Delta, H \vdash th : \text{Thread} \). Since \( H \) and \( V S_{th'} \) are not changed by E-MethodVoidStart for threads other than \( th \), we derive \( \Delta, H \vdash V S_{th'} : \text{ok} \) from \( \Delta, H \vdash TP : \tau \). For \( \Delta, H \vdash (BS_{th} \circ [] \circ [] \circ \text{ok}) \), it is enough to show that \( \Delta, H \vdash th : C \), which is immediate. For every thread \( th' \) including \( th \), we should prove (48) \( \Delta, H, V S_{th}, \Phi_{th} \vdash CF_{th} \circ FS_{th} : \text{void} \rightarrow \tau' \) such that \( \tau' \prec \tau \), where \( \tau \) is given by the assumption (49) \( \Delta, H, V S_{th}, \Phi_{th} \vdash CF_{th} \circ FS_{th} : \text{void} \rightarrow \tau \). For threads other than \( th' \) and \( th \), \( CF_{th} \) and \( FS_{th} \) are not changed by this transition. Moreover, we have \( \Phi_{th} \subseteq \Phi_{th} \cup \{ th \} \) for such threads because \( V S_{th} \) and \( H \) do not change. Note that there is no variable or field with the value of \( th \) in the scope of these threads. This is due to Corollary 1 and Lemma 4. For the same reason, \( th \) does not appear in the closed frame and frame stack of threads other than \( th' \) and \( th \). For \( th' \), according to TFF-StackClosed and by assuming (49), we have (50) \( \Delta; \text{context}(H, V S_{th'}); \Phi_{th} \vdash th_.\text{start}() : \text{void} \). From (51) and \( \Phi_{th} \subseteq \Phi_{th} \), it can be concluded that (52) \( \Delta, H, V S_{th}, \Phi_{th} \vdash F \circ FS_{th} : \text{void} \rightarrow \tau \). This holds because \( \Phi_{th} \) is composed of \( \Phi_{th} \), \( th \), and those expressions in \( s_{th} \), on which \( \text{start} \) is invoked. Therefore, (48) is satisfied for \( th' \). For \( th \), we should prove (53) \( \Delta, H, V S_{th}, \Phi_{th} \vdash (\text{return } th_.) \circ [] \circ \text{void} \rightarrow \tau \). To do so,
it is enough to show that (54) \( \Delta; \text{context}(H, VS_{th}); \Phi \vdash s_{th}; \text{return} \; th: \; C[\Phi'] \)

is derivable. Then, a reasoning similar to what given in E-New can be applied to obtain (53). We have \( \Phi_{th}^2 = \text{dom}(TP_2) \cup \text{giveThreads}(H, VS_{th}, \text{dom}(TP_2)) \), where the set that is returned by \( \text{giveThreads} \) includes \( \text{this} \). It may also include \( \text{this}.f_1.f_2. \cdots .f_n \) for a sequence of field references \( f_1, f_2, \cdots , f_n \) if the value of \( th.f_1.f_2. \cdots .f_n \) is in \( \text{dom}(TP_1) \). Note that, form Lemma 3, the value of this expression cannot be \( th \) itself. The set \( \Phi \) against which the body of \( \text{run} \) in (50) is type checked is equal to \( \Phi_{th}^1 \cup \{ th \} \). An argument similar to the one given for E-Method can then be given to deduce (54). That is, the body of \( \text{run} \) against an empty \( \Phi \) is typable on the basis of T-MDefn. Moreover, it is typable against \( \Phi_{th}^1 \cup \{ th \} \) which includes \( \text{dom}(TP_1) \). The variable \( \text{this} \) cannot be a member of \( \Phi_m \) because \( \Phi_m \) in which \( th \) is substituted for \( \text{this} \) is type checked successfully against \( \Phi_{th}^1 \cup \{ th \} \). Furthermore, if \( \text{this}.f_1.f_2. \cdots .f_n \) is in \( \Phi_m \), the body is typable against \( \Phi_{th}^2 \). In this case, such an expression cannot be a member of \( \Phi_{th}^1 \). This is because \( th.f_1.f_2. \cdots .f_n \) as a member of \( \Phi_m \) is successfully checked against \( \Phi_{th}^1 \cup \{ th \} \) which does not include the value of \( th.f_1.f_2. \cdots .f_n \). In fact, \( th \) and \( \text{this} \) as the corresponding pair of actual/formal parameters is well handled by TE-METHOD.

- (E-DeleteThread) As \( TP_1 \) is well-typed, the premises of TG-TP are satisfied. Thus, \( TP_2 \) is well-typed as well. Note that since \( \text{dom}(TP_2) \subseteq \text{dom}(TP_1) \), we have \( \Phi_{th}^2 \subseteq \Phi_{th}^1 \) for every thread \( th \in \text{dom}(TP_2) \).