Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

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

Zeinab Iranmanesh[a] Mehran S. Fallah[ab]

a. Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), P.O. Box: 15875-4413, Tehran, Iran

b. School of Computer Science, Institute for Research in Fundamental Sciences (IPM), P.O. Box: 19395-5746, Tehran, Iran

Abstract We present MTMJ, a multithreaded middleweight model language for Java which is strongly typed and prevents multiple run-time start of threads. The key point in designing the language is to balance precision and efficiency by judicious utilization of context information during type checking. While the types are flow-insensitive, the required flow-dependent information is collected as type checking progresses. We prove that our type system is sound and guarantees the good behavior of well-typed programs. In particular, the execution of a typable program does not lead to multiple start of threads. We also develop a type checker as part of this research and apply it to several MTMJ programs.

Keywords Java, Multiple Start of Threads, Multithreading, Type Safety

## 1 Introduction

Multithreading is a feature of the Java programming language [GJSB05] by which an intended functionality can be implemented through a number of concurrent threads. A multithreaded system developed in this way, however, may be of high complexity so that it may not be easily assured that the system behaves well at run-time. The property that every syntactically correct program is well-behaved is known as safety and can be brought to the language through static or dynamic checking of programs. The study of safety in a full-blown language like Java, however, is an arduous task. Thus, one should first design and study a model language reflecting some, but not all, features of the language it models. The results can then be used to retrofit the language under study.

There are a number of model languages for Java among which Middleweight Java [BPP03], MJ for short, has a fairly rich set of features making it analogous to Java while remaining amenable to in-depth analysis. MJ is a proper subset of Java in the sense that its programs can be compiled and executed as ordinary Java programs. In this paper, we concentrate on multithreading and the way it interacts with the other features of Java. In doing so, we propose MTMJ which is a multithreaded language based on MJ, hence its name. MTMJ prevents bad behaviors through its type system, i.e., it is type-safe. In effect, it fulfills some new requirements for safe multithreading. Therefore, a typable program will not enter a state at which no semantic rule can be applied and, in turn, no transition to other states is possible. In this way, it is guaranteed that well-typed programs do not encounter any unexpected error at run-time.

The principal feature of MTMJ is multithreading. To create a thread in Java, one should instantiate the class **Thread**. By invoking the method **start** on the instance, then, the run-time system starts scheduling and executing that thread. These constructs are incorporated into MTMJ without change. To study the safety of MTMJ, thus, it is reasonable to examine those errors that may occur as a result of using such constructs. A number of concurrency errors in Java, such as those occurred due to data races, have been extensively studied in the literature, e.g., [SAWS05, DP09, WCG11, FFLQ08, MPA05].

In this paper, we concentrate on multiple invocation of **start** on the same **Thread** object. It could simply occur when a programmer believes wrongly that he can restart a thread by invoking **start** on the corresponding object. Consider a variable `th` on which **start** has already been invoked representing a thread which performs some task. It could be very common to invoke **start** again on `th` to repeat the corresponding task. There are reports of software crashes resulted from such an illusion, e.g., [Bug12b, Bug12a]. The run-time exception corresponding to this error has been reported in the issue tracker of Android open-source project [And08]. It should be noted that since such an error involves pointers around the whole program, programmers may get in trouble to detect and remove the source of inadvertent multiple invocation of **start**.

The rationale behind double invocation of **start** on the same thread is, in general, to restart that thread or to start a new thread while the previous one is executing. The following argument explains why the things that may happen as a result of double invocation of **start** differ from what a programmer intends, and in turn, why we should prevent it in a safe language. The argument is based on the fact that the interaction with a running thread is possible only through the methods of the object representing that thread, i.e., the object is a handle for the running thread.

An interpretation of restarting a thread is to stop that thread, if it is running now, and to reschedule its whole instructions for a new execution. To realize such an interpretation, the **Thread** object started earlier should be the handle of the new thread created by the second invocation of **start**. However, as that object conveys the state of the previous thread, this may lead to a misconception about the actual state of the new thread. This endangers safety because the integrity of the high-level abstractions introduced by the programmer through invoking **start** on a thread is not guaranteed under such an interpretation.

Another interpretation of invoking **start** on an object whose corresponding thread is running now is to create a second thread such that both threads are referred to by the same object at the same time. By such an interpretation, a programmer will not be certain which thread is affected by the methods he invokes on the object. It is not clear, for example, how the result of invoking **isAlive** on the object can be

```
Thread1 th; //Thread1 is a subclass of Thread
th = new Thread1();
th.start();
(Thread)th.start();
```

(A)

```
Cell y; //the class Cell has a field f of type Thread
y = new Cell();
y.f = new Thread();
y.f.start();
Cell x;
x = y;
x.f.start();
```

(B)

Figure 1 – Example ill-typed programs containing the kind of aliasing not allowed in MTMJ.

interpreted. In effect, the same resource—a **Thread** object—can be simultaneously accessed by different processes in a possibly wrong manner.

The current implementations of Java deal with multiple invocation of **start** on the same object in two ways: to raise a run-time exception [Ora11] or to ignore any subsequent invocation of **start** [Nau96] altogether. Such an equivocal treatment of this error confuses programmers further; example evidences are the bugs reported to the bug database of Oracle [Ora05, Ora02]. Without a clear specification of the cases handled by each solution, programmers may not even become aware of the existence of errors in their programs—they may expect an exception if the program leads to multiple start of threads but the run-time system takes the other way.

Inspired by the existing research on static detection of run-time errors such as dereferencing null pointers and exceeding array bounds, e.g., [Saw98, FLL$^+$02, HSP05, PAC$^+$08, DDE$^+$11], MTMJ prevents multiple invocation of **start** on the same object statically. To the best of our knowledge, the only reported result in static detection of this error is the tool called E_Jlin [HAT$^+$04]. Apart from being exogenous to the language, it is not guaranteed that E_Jlin can certainly detect the error. The static solution of MTMJ outperforms the run-time solution of Java provided the cost and limitations it imposes are acceptable. In effect, identifying such an error only on the basis of the text of a program may require complicated and costly analyses. In particular, the expressions of type **Thread** that point to the same object—aliases for expressions of type **Thread**—should be identified and stored during type checking. The existence of several forms of aliasing in MTMJ will aggravate the problem.

MTMJ controls aliasing in such a way that each thread identifier can be the value of only one expression of type **Thread**. This restriction has little impact on the expressiveness of the language as the main reason to have expressions of type **Thread** is to start corresponding threads somewhere in the program, and thus, aliases for a thread are likely to lead to multiple start of the same thread. Nevertheless, to alleviate the impact of such decisions on correct programs, we give guidelines to convert an ill-typed, but correct, program to a well-typed program (See Table 1). The programs in Fig. 1 contain the aliasing that is not allowed in MTMJ. In Program A, the expression "`(Thread)th`" of type `Thread` is an alias for the expression "`th`" of type `Thread1`, a subclass of `Thread`. The assignment of "`y`" to "`x`" in Program B is unsafe as well. In fact, the expression "`y.f`" is an alias for "`x.f`" and both expressions are of type `Thread`.

The types of MTMJ are not flow-sensitive [FTA02, Pea10]. That is, the type of the same expression does not depend on the place it appears in the program. Nonetheless, MTMJ's type system indeed emulates flow-sensitive typing using the sets in the typing context that are modified during type checking—it is a type and effect system. It should be noted that some flow-sensitive static analysis methods have been devised based on a number of type reconstruction rules similar to the typing rules of MTMJ. Another point about the type system of MTMJ is that it does not require any additional type annotation for the analysis of multiple start of threads. We prove that the type-system of MTMJ is sound. A type checker is also implemented that automates the analysis of MTMJ programs. Finally, it should be noted that the type-based program analysis proposed in this paper is modular. In effect, a program is well-typed if its methods are well-typed. A method itself is well-typed if the statements constituting its body are type-checked successfully. Moreover, the type of a statement or expression depends only on the types of that statement's or expression's components.

As MTMJ is composed of a fairly rich set of Java features, its type system can be applied to many Java programs. Moreover, the type system of Java can be modified according to the typing rules of MTMJ. This is among the topics for future research. The manner in which MTMJ prevents multiple start of threads may also be used to statically prevent some other similar errors such as the one that may be raised when the method **setDaemon** in the class **Thread** of Java is invoked. Note that multiple start of threads is not specific to the Java programming language. It may occur in any object-oriented language where a thread is created through the invocation of a similar method on a thread object. Therefore, the constraints imposed by the static semantics of MTMJ may also prevent erroneous start of threads in other programming languages such as C♯ [Mic01] and Python [Pyt15]. It should be noted that although the Java programming language provides high-level abstractions for handling threads, we decide on **Thread**, which is the base class for multithreading in Java. By preventing the errors regarding this construct, it can be used safely besides other higher-level classes provided for concurrent programming.

This paper goes on as follows: Related works are discussed in Section 2. Section 3 gives the syntax and static semantics of MTMJ. Section 4 presents the dynamic semantics of MTMJ. We show that MTMJ is strongly typed and it guarantees at most one invocation of **start** on any **Thread** object. The proof sketch is in Section 5—details can be found through the web page containing the codes, software, and some other things developed as part of this research [ISF16a]. Finally, Section 6 concludes the paper.

## 2 Related Work

### 2.1 Models for Java

There are a number of model languages for Java. Featherweight Java (FJ) [IPW01] is a purely functional language in which no side effect occurs when an expression is evaluated. Thus, it is not a good model for Java where changes in observable states underlie the execution of a program. Classic Java [FKF99] may also be considered as a model language for Java, although it contains some syntactic constructs, e.g., let binding, not appearing in Java. In fact, it is not a subset of Java. Middleweight Java [BPP03] is an imperative subset of Java with an appropriate set of features. There are also smaller subsets of Java such as Lightweight Java (LJ) [Str10] and larger subsets like the one introduced by Drossopoulou et al. [DEK99, DVE00].

Multithreading has also been incorporated into some models of Java. For instance, Concurrent Java [FF00] is an extension of Classic Java that supports multithreading. Its expression "**fork** $e$" spawns a new thread whose body is $e$. This expression is then evaluated only for its side effects. As with Classic Java, Concurrent Java is not a subset of Java. Welterweight Java (WJ) [OW10] is another example based on FJ which is imperative and supports concurrency. There are model languages that include some other features of Java. For example, remote method invocation has been considered in [AY07]. As another example, the language presented in [BDSS10] provides code reuse through the units named records and traits.

Some core calculi have been proposed for object-oriented programming. For example, the object calculus [AC96] is a simple, yet powerful, mathematical model for the analysis of object-based languages. Moreover, the concurrent object calculus [GH99] supports multithreading. A core calculus based on the lambda-calculus has also been proposed for class-based languages [BPSM99]. Evidently, model languages like MTMJ are less abstract than core calculi, thereby being more suitable for the analysis of real-life languages such as Java.

## 2.2 Type Systems

Type systems have long been recognized as an effective mechanism to statically detect program errors. In particular, standard type systems for mainstream object-oriented languages such as C++, C♯, and Java ensure that well-typed programs behave well in many respects. For example, they typically guarantee that all references to the attributes—fields and methods—of any object are correct in the sense that the class of the object or one of its direct or indirect superclasses contains the field or the method being referred and the types of actual and formal parameters to methods are consistent. Notwithstanding, such type systems usually cannot prevent the run-time errors having complicated syntactic patterns. An example is null pointer dereferencing.

The point stated above has motivated scholars to propose more sophisticated type systems. To provide memory safety for X10 programs [CGS$^+$05], for example, the basic type system of the language has been extended with region types [Gro06]. Constrained types, which are the basic types associated with predicates on the object's state, have also been introduced to X10 [NSPG08]. By these types, one can statically detect the errors that may occur due to inconsistent design requirements of program components. As another example, session types have been used as a means to prevent deadlock during sessions where a session is defined to be a sequence of interactions between two threads [DCMYD06]. Type systems are also used to enforce security properties. For example, a security type system is proposed in [ISF16b] which provably enforces a noninterference property defining allowable executions which do not lead to insecure information flows. As with the attempts briefly reviewed above, MTMJ aims at adding a new capability to the type systems of object-oriented languages. The type system of MTMJ prevents multiple start of threads, an error resulting from multithreading in object-oriented languages.

## 2.3 Multiple Start of Threads

There are some works that deal with double invocation of **start** on the same object. In Welterweight Java, it is resolved through the dynamic semantics where a new thread object is instantiated for each run-time invocation of **start**. This solution, however, has its own drawbacks. First, it requires some changes in the syntax of Java. For example, **start** is invoked on a class name instead of an object of type **Thread**. Second, such a solution may result in an execution which differs from what a programmer expects. Another approach in preventing erroneous **start** invocation is program testing [HAT$^+$04].

The problem of aliasing, which can be thought of as a source of erroneous **start** invocation, has been studied extensively in the literature, e.g., [Wad90, Min96, CPN98, AKC02, Boy04, KA08, Die09]. Some works make use of linear types [Wad90] which guarantee that a resource is used at most once. So-called unique pointers [Min96] which cannot point to shared memory cells are built on the concept of linear types. The implementation involves destructive reads which set the value of a unique variable to null once the read occurs [Hog91, Min96]. The main drawback of this approach is that programmers may not be aware of the side effects of setting a unique variable to null. Although such a uniqueness is provided in [Boy01] without using destructive reads, the static analysis method underlying the solution is highly sophisticated. Moreover, linear methods which may be invoked once are introduced in [KA08] where methods are associated with objects. In the context of class-based object-oriented languages with methods linked to classes instead of objects, our approach is more flexible in the sense that **start** can be invoked many times, but each time on a new **Thread** object.

Some other works utilize ownership types [CPN98, CÖSW13] to place restriction on access to shared objects. Early ownership types are too restrictive and substantially reduce the expressiveness of the language. There are some more flexible variants as well [BV99, MPH99]. The achievements are interesting, but they require programmer to annotate types. The type system we have proposed in this paper prevents some kinds of aliasing without the need for additional type annotations. Furthermore, we have investigated all possible constructs that may lead to aliasing in a powerful subset of Java. The results can thus be employed in concurrency control, memory management, and security in other languages where such constructs or their equivalents exist. It should be noted that the researches performed on tracking typestates such as [DF04, ASSS09] can be applied to prevent double start of threads, but they also face the overhead and complexity imposed by requiring several annotations.

## 2.4 Graph-based Program Analysis

In addition to type systems, some other static methods are proposed for program analysis which are based on program dependence graphs (PDG). The PDG [FOW87] of a program is a graph that illustrates data and control dependencies among the statements and expressions of the program—program dependence graphs and control flow graphs may be used interchangeably. Such an analysis is automated by tools like FindBugs [The15]. This tool checks programs for the absence of run-time errors, e.g., the direct invocation of **run** instead of the invocation of **start** to create threads. ThreadSafe [AS15] is another tool for detecting some concurrency-related errors which are mainly categorized into race conditions, deadlocks, unpredictable results, and performance bottlenecks. These tools, however, do not detect multiple start of threads.

| | | | |
|---|---|---|---|
| $P$ | $::=$ | $cd_1 \ldots cd_n ; \bar{s}$ | **Program** |
| $cd$ | $::=$ | $\textbf{class } C \textbf{ extends } C \ \{fd_1 \ldots fd_k \ cnd \ md_1 \ldots md_n\}$ | **Class Definition** |
| $fd$ | $::=$ | $C \ f;$ | **Field Definition** |
| $cnd$ | $::=$ | $C(C_1 x_1, \ldots, C_j x_j)\{\textbf{super}(e_1, \ldots, e_k); \ s_1 \ldots s_n\}$ | **Constructor Definition** |
| $md$ | $::=$ | $\tau \ m(C_1 x_1, \ldots, C_j x_j)\{s_1 \ldots s_n\}$ | **Method Definition** |
| $\tau$ | $::=$ | $C \mid \textbf{void}$ | **Return Type** |
| $e$ | $::=$ | $x \mid \textbf{null} \mid e.f \mid (C)e \mid pe$ | **Expression** |
| $pe$ | $::=$ | $e.m(e_1, \ldots, e_k) \mid \textbf{new } C(e_1, \ldots, e_k)$ | **Promotable Expression** |
| $s$ | $::=$ | $; \mid pe; \mid e_1.f = e_2; \mid C \ x; \mid x = e;$ | **Statement** |
| | $\mid$ | $\textbf{return } e; \mid \{s_1 \ldots s_n\}$ | |
| | $\mid$ | $\textbf{if } (e_1 == e_2) \ \{s_1 \ldots s_k\} \textbf{ else } \{s_{k+1} \ldots s_n\}$ | |

Figure 2 – The syntax of MTMJ.

These analysis tools are not sound either. It is worthy of mention that a type-based analysis is compositional in the sense that the modular analysis of different program components can be combined to form the whole analysis. Moreover, such an analysis is more declarative than graph-based analysis methods, which are mainly algorithmic.

## 3 Syntax and Static Semantics

### 3.1 Syntax

The syntax of MTMJ is given in Fig. 2. As can be seen, an MTMJ program is a collection of class definitions together with a sequence of statements $\bar{s}$ representing the main body of the program. In addition to the classes defined by the programmer as well as the built-in class **Object**, we consider a new built-in class **Thread** which extends **Object** and represents threads. This class is the change made to the syntax of MJ. The body of **Thread** is considered to be composed of a constructor and the two methods **run** and **start** with the return type **void** and no method parameters. For the sake of simplicity, we assume that the same local variables do not appear in methods with different names. It is worthy of mention that although the syntax does not contain "for" and "while" loops, the same functionality can be modeled using recursive method invocation.

To create a new thread, we define a subclass of **Thread**. Then, we override the method **run** with new instructions and instantiate an object of this subclass. When invoked on this object, the method **start** allocates the required resources and the thread is scheduled for execution. It is assumed that the allocation of resources is always performed successfully and that there is no way to override **start**.

### 3.2 Basics of the Type System

Class names are considered as types. Moreover, the type information of classes is obtained from a class table which is considered as a fixed part of any typing context. A class table $\Delta$ is a triple of functions $(\Delta_m, \Delta_c, \Delta_f)$. The type of a method $m'$ of a class $C$ is

$$\Delta_m(C)(m') = C_1, ..., C_n \prec\!\!\langle \Phi_{m'} \rangle\!\!\rightarrowtail \tau,$$

where $C_i$ and $\tau$ are the type of the $i$th parameter and the return type of $m'$, respectively. The set $\Phi_{m'}$ comprises the expressions on which **start** is invoked in the body of $m'$. Note that we use **void** as input type when there is no input parameter to the method and as output type if the body of the method does not return any value. Similarly, $\Delta_c(C)$ is a sequence $C_1, ..., C_n, \Phi_C$, where $C_i$ is the type of the $i$th parameter to the constructor of class $C$ and $\Phi_C$ is composed of the expressions in the constructor on which **start** is invoked. The type of a field $f'$ of class $C$ is $\Delta_f(C)(f')$ as well. Moreover, the subclass relation "$\prec$" and the direct subclass relation "$\prec_1$" are defined as in MJ.

It should be noted that the set $\Phi_m$ in the type of method $m$ contains the elements of $\Phi_{m'}$ of any method $m'$ invoked in the body of $m$. The typing rule concerning the invocation of method $m$, then, utilizes $\Phi_m$ to prevent any illegal invocation of **start**. It is assumed that the body of $m$ can be overridden in subclasses provided $\Phi_m$ remains a subset of that in the superclass. This facilitates the static analysis of programs in the presence of dynamic dispatch. In fact, by this restriction, it is enough to only investigate $\Phi_m$ in the superclass and there is no need to check for the invocation of **start** in the body of overridden methods. A correct program that is deemed ill-typed due to noncompliance with this restriction can be converted to a typable program as shown in the fourth row of Table 1. In effect, a conditional structure, whose condition always evaluates to true and **else** branch contains the extra members of $\Phi_m$ in the subclasses, is added to the body of $m$ in the superclass. This table also contains guidelines regarding other restrictions that may cause a correct program to be regarded as an ill-typed program.

At the beginning of type checking, $\Delta_m$, $\Delta_c$, and $\Delta_f$ are initialized as functions shown in Fig. 3. The sets $\Phi_m$ and $\Phi_C$ are initially undefined but they are computed as type checking progresses. The function $mbody(C, m)$ returns the arguments and the body of $m$ when it is considered as a method in class $C$. The same information is obtained from the function $cnbody(C)$ for the constructor of $C$.

The type system of MTMJ is given in Figs. 5-9. We present only some more intricate typing rules in order to convey the main concepts of the type system. The complete set of rules can be found in the online appendix of this paper [ISF16a]. As with Java and as justified in Section 1, the method **start** must not be invoked more than once on an object of class **Thread** or its subclasses. The typing rules of MTMJ prevent such an error.

The typing judgment has two sets $\Phi$ and $\Phi'$, with $\Phi \subseteq \Phi'$, in its hypotheses and consequent, respectively. As expressions evaluate to object identifiers, by invoking a method on an expression, we mean the execution of the method identified by the value—object identifier—of that expression. During type checking, the sets $\Phi$ and $\Phi'$ keep track of those expressions of type **Thread** on which **start** has already been invoked. That is, the type system predicts the run-time invocation of **start** on such expressions in the execution of part of the program type checked so far. The expressions that begin with **new**, however, are not collected in $\Phi$, since their evaluation culminates in a new thread on which **start** has not been invoked yet—the type system, indeed, does not allow the corresponding thread to start before being returned by the class constructor.

A double invocation of **start** on an expression in $\Phi$ is prohibited by the type system of MTMJ. The invocation of **start** on an expression that evaluates to the

Table 1 – Guidelines to convert ill-typed correct programs to well-typed programs.

| The phrase and the reason for being ill-typed | A well-typed substitute |
|---|---|
| $(C')e$, aliasing through up-casting a thread expression $e$ of type $C$ | 1. Declare a local variable of type $C'$ using $C'$ $x$;<br>2. Instantiate $x$ as a value of type $C$ by $x$ = **new** $C()$;<br>3. Set the field $f$ of $x$ to that of $e$ using $x.f = e.f$; if $f$ doe not contain any reference to a **Thread** object<br>4. Instantiate other fields $f$ of $x$ to values of corresponding types $C_f$ using $x.f$ = **new** $C_f()$; and repeat (3) and (4) for the fields of $x.f$ |
| $e.m(e_1, \ldots, e_n)$, aliasing through invoking method $m$ whose return type is $C$ and returns a **Thread** object or an object containing a reference to a **Thread** object | 1. Declare a local variable of type $C$ through $C$ $x$;<br>2. Instantiate $x$ as a value of the same type using $x$ = **new** $C()$;<br>3. Add a fresh formal parameter $y$ of type $C$ to $m$<br>4. Pass $x$ as an actual parameter to $m$ corresponding to $y$ by $e.m(e_1, \ldots, e_n, x)$<br>5. Remove the last statement in the body of $m$ which is **return** $e'$;<br>6. Set the return type of $m$ to **void**<br>7. If $e'$ is not **new** $C()$, set the fields of $y$ to those of $e'$ at the end of the body of $m$ similar to what explained in the case of up-casting |
| $e.f = e'$;, aliasing through assigning a thread expression of type $C$ to a field where the expression is not **null** and does not begin with **new** | 1. Instantiate $e.f$ as a value of type $C$ through $e.f$ = **new** $C()$;<br>2. Set the fields of $e.f$ to those of $e'$ through the steps explained in the case of up-casting |
| Method $m$ in class $C$ is overridden in a subclass $C'$ such that there are new invocations of **start** in the overridden method | 1. Add a conditional structure **if**(**null** == **null**){}**else**{} to the body of $m$ in $C$<br>2. Add extra invocations of **start** of $m$ in $C'$ to the else branch of the structure above |
| Method $m$ is in a loop of invocations and there is an expression $e$.**start**() in $m$ where $e$ is of type $C$, does not begin with **new**, and contains some formal parameters to $m$ | 1. Declare a local variable of type $C$ using $C$ $x$;<br>2. Instantiate $x$ as a value of the same type<br>3. Set the fields of $x$ to those of $e$ through the steps explained in the case of up-casting<br>4. Invoke **start** on $x$ |

$$\Delta_m(C)(m') = \begin{cases} \bar{C} \prec\!\langle\uparrow\rangle\!\succ\!\rightarrow \tau \\ \qquad \textit{if } P = *\ \textbf{class}\ C\ \textbf{extends}\ C'\{*\ \textit{cnd}\ md_1 \ldots md_n\}\ * \\ \qquad \wedge\, \exists 1 \leq i \leq n.\ md_i = \tau\ m'(\bar{C}\ \bar{x})\{\bar{s}\} \\ \\ \Delta_m(C')(m') \\ \qquad \textit{if } P = *\ \textbf{class}\ C\ \textbf{extends}\ C'\{*\ \textit{cnd}\ md_1 \ldots md_n\}\ * \\ \qquad \wedge\, \forall 1 \leq i \leq n.\ md_i \neq \tau\ m'(\bar{C}\ \bar{x})\{\bar{s}\} \end{cases}$$

$$\begin{aligned} \Delta_c(C) = {} & C_1, \ldots, C_j, \uparrow \\ & \quad \textit{if } P = *\ \textbf{class}\ C\ \textbf{extends}\ C'\{*\ \textit{cnd}\ *\}\ * \\ & \quad \wedge\, \textit{cnd} = C(C_1\ x_1, \ldots, C_j\ x_j)\{\bar{s}\} \end{aligned}$$

$$\Delta_f(C)(f') = \begin{cases} C'' \\ \qquad \textit{if } P = *\ \textbf{class}\ C\ \textbf{extends}\ C'\{fd_1 \ldots fd_n *\}\ * \\ \qquad \wedge\, \exists 1 \leq i \leq n.\ fd_i = C''\ f';\ \wedge\, \Delta_f(C')(f') \uparrow \\ \\ \Delta_f(C')(f') \\ \qquad \textit{if } P = *\ \textbf{class}\ C\ \textbf{extends}\ C'\{fd_1 \ldots fd_n *\}\ * \\ \qquad \wedge\, \forall 1 \leq i \leq n.\ fd_i \neq *\ f'; \end{cases}$$

Figure 3 – Initialization of the class table. The symbol "$*$" stands for wild card and "$\uparrow$" for undefined.

value of a member of $\Phi$ is prevented as well. To identify such expressions, we may decide to store all assignments to expressions of type **Thread**. However, this solution is too expensive and may render type checking a costly process. As a solution to this problem, we prevent two different expressions from pointing to the same object of type **Thread**—such an object may be a field of an object, a field of a field of an object, and so on. In this way, we can prevent the bad aliasing stated above but at the cost of preventing some correct phrases. This constraint does not reduce the expressiveness of MTMJ. As far as we have investigated, the programmer can compensate for the above restriction in safe programs, i.e., the programs without erroneous double invocation of **start**, by creating appropriate new objects, for example. We will expound on this issue when explaining our typing rules.

It is assumed that if some invocation of **start** such as $e$.**start**() occurs in a recursive method $m$, the expression $e$ either begins with **new** or does not contain any parameter of $m$. By this assumption, $\Phi_m$ is derived by one-pass checking of the body of $m$. Otherwise, $\Phi_m$ may include infinitely many expressions making type derivation undecidable. For example, consider the following code where class `C` has a field `f` of type **Thread** and a field `g` of type `C`. As stated earlier, in every run of $m$, the expression `new Thread()` is guaranteed to return a new thread on which **start** has not been invoked yet. Considering `x.f.start()`, however, the set $\Phi_m$ contains `x.f`, `x.g.f`, `x.g.g.f`, and so on. This infinite set is obtained because `x` is replaced with `x.g` each time the method is invoked recursively.

```
void m(C x) {
    new Thread().start();
    x.f.start();
    this.m(x.g);
}
```

It is worthy of mention that if the above assumption is not true for some recursive method, such a method can be converted to a typable one as shown in the last row of Table 1.

$$\Delta_{th}(C) \quad = \quad \begin{cases} \mathit{true} & \mathit{if}\, C \prec \textbf{Thread} \\ \mathit{true} & \mathit{if}\, \exists f, C'.\ \big(C' = \Delta_f(C)(f) \wedge \Delta_{th}(C')\big) \\ \mathit{true} & \mathit{if}\, \exists C''.\,(C'' \prec C \wedge \Delta_{th}(C'')) \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

$$\mathit{giveMethod}(C, m) \quad = \quad \begin{cases} \textbf{run} & \mathit{if}\, C \prec \textbf{Thread} \wedge m = \textbf{start} \\ m & \mathit{otherwise} \end{cases}$$

$$\mathit{giveStart}(C, m, e) \quad = \quad \begin{cases} \{e\} & \mathit{if}\, C \prec \textbf{Thread} \wedge m = \textbf{start} \wedge e \neq \textbf{new}\ * \\ \emptyset & \mathit{otherwise} \end{cases}$$

$$\begin{array}{r} \mathit{checkIfStart} \\ (m, C, e, \Phi) \end{array} \quad = \quad \begin{cases} \mathit{true} & \mathit{if}\, m \neq \textbf{start} \\ \mathit{true} & \mathit{if}\, C \not\prec \textbf{Thread} \\ \mathit{true} & \mathit{if}\, e \notin \Phi \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

$$\begin{array}{r} \mathit{replace}(x_1, \ldots, x_n, \\ \Phi, e'_1, \ldots, e'_n) \end{array} \quad = \quad \begin{array}{l} \{e'' | \exists e \in \Phi.\ \big(e'' = \big[e'_i/^n_{i=1} x_i\big]\, e \wedge e'' \neq \textbf{new}\ *\big)\,\} \\ \mathit{if\ not}\ \exists e, e' \in \Phi.\ \big(e \neq e' \wedge \big[e'_i/^n_{i=1} x_i\big]\, e = \big[e'_i/^n_{i=1} x_i\big]\, e' \neq \textbf{new}\ *\big) \end{array}$$

$$\mathit{check}(\Phi', \Phi) \quad = \quad \begin{cases} \mathit{false} & \mathit{if}\, \exists e'.\ e' \in \Phi' \cap \Phi \\ \mathit{true} & \mathit{otherwise} \end{cases}$$

$$\mathit{chTP}(\Delta, m, C) \quad = \quad \begin{cases} \mathit{true} & \mathit{if}\ \exists C'.\,(C \prec_1 C' \wedge m \notin \mathit{dom}(\Delta_m(C'))) \\ \mathit{true} & \mathit{if}\ \exists C'.\ \left( \begin{array}{l} C \prec_1 C' \wedge m \in \mathit{dom}(\Delta_m(C')) \wedge \\ \Delta_m(C)(m) = C_1, \ldots, C_n \prec\!\!\{\Phi^C_m\}\!\!\succ\!\!\to \tau \wedge \\ \Delta_m(C')(m) = C_1, \ldots, C_n \prec\!\!\{\Phi^{C'}_m\}\!\!\succ\!\!\to \tau' \wedge \\ \phi^C_m \subseteq \Phi^{C'}_m \wedge (\tau = \tau' \vee \tau \prec \tau') \end{array} \right) \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

Figure 4 – Definitions used in the typing rules of MTMJ. The symbol "∗" stands for wild card and $P$ for the program being type checked.

## 3.3 Typing Rules for Expressions

The typing rules of MTMJ for some expressions are given in Fig. 5. By a set being undefined, we mean that it has not been derived yet in the process of type checking—a set which has been derived is, therefore, a defined set.

### 3.3.1 The Rule for Casting

The rule TE-UpCast copies the set $\Phi'$ from its premise into its conclusion. The judgment $\Delta \vdash C'\ ok$ states that $C'$ is in the domain of $\Delta$. Moreover, this rule has 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 $e$—the definition of $\Delta_{th}(C)$ is given in Fig. 4. 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.

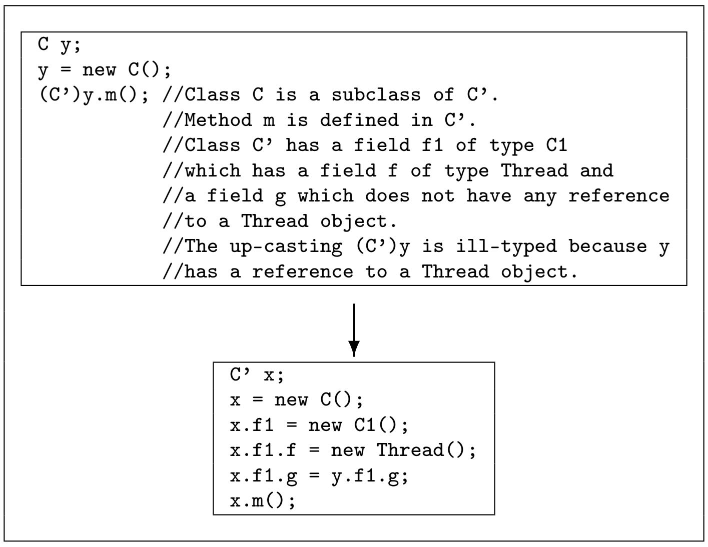
If the casting $(C')e$ is ill-typed because $\neg\Delta_{th}(C)$ does not hold where $C$ is the class of $e$, we show that there is a typable code that can emulate what is intended from such a casting. As seen in Fig. 4, $C \prec C' \prec$ **Thread** is the first case which causes $\Delta_{th}(C)$ to be true. In such a case, the substitute code contains a declaration of the form "$C'\ x = \textbf{new}\ C();$" that defines an expression of type $C'$, i.e., the type of

$$\frac{\Delta;\Gamma;\Phi\vdash e:C|\Phi' \quad C\prec C' \quad \Delta\vdash C'\ ok \quad \neg\Delta_{th}(C)}{\Delta;\Gamma;\Phi\vdash (C')e:C'|\Phi'}$$
$$\text{(TE-UpCast)}$$

$$\frac{\begin{array}{c}
\Delta;\Gamma;\Phi\vdash e:C_0|\Phi_0 \quad \Delta;\Gamma;\Phi_0\vdash e_1:C_1|\Phi_1\ \ldots\ \Delta;\Gamma;\Phi_{n-1}\vdash e_n:C_n|\Phi_n\\
m_g = \mathit{giveMethod}(C_0,m) \quad \Delta_m(C_0)(m_g)=C'_1,\ldots,C'_n \prec\!\langle\Phi_{m_g}\rangle\!\!\rightarrowtail \tau\\
C_1\prec C'_1\ \ldots\ C_n\prec C'_n \quad \mathit{mbody}(C_0,m_g)=(x_1,\ldots,x_n,\bar{s})\\
\mathit{checkIfStart}(m,C_0,e,\Phi_n) \quad \Phi_{current}=\mathit{giveStart}(C_0,m,e)\\
\mathit{if}\ \ \Phi_{m_g}\uparrow\ \ \mathit{then}\\
\left(\begin{array}{c}
\mathit{if}\ \ \Delta;(C_0,m_g)\vdash \mathit{loop}(C_0,m_g)\\
(\mathit{then}\ \ \Delta_1=\Delta[\Delta_m(C_0)(m_g)\mapsto C'_1,\ldots,C'_n \prec\!\langle\emptyset\rangle\!\!\rightarrowtail\tau]\ \ \mathit{else}\ \ \Delta_1=\Delta)\\
\Gamma_1 = x_1:C'_1,\ldots,x_n:C'_n,\mathbf{this}:C_0 \quad \Delta_1;\Gamma_1;\emptyset\vdash\bar{s}:\tau|\Phi''
\end{array}\right)\\
\mathit{else}\ \ \Phi''=\Phi_{m_g}\\
\Phi'_{m_g} = \mathit{replace}(\mathbf{this},x_1,\ldots,x_n,\Phi'',e,e_1,\ldots,e_n)\wedge\Phi'_{m_g}\downarrow\\
\mathit{if}\ \ \Phi'_{m_g}\neq\emptyset\ \ \mathit{then}\ \ \mathit{check}(\Phi'_{m_g},\Phi_n\cup\Phi_{current})\\
\Phi'=\Phi_n\cup\Phi_{current}\cup\Phi'_{m_g}\\
\mathit{if}\ \ \tau\neq\mathbf{void}\ \ \mathit{then}\ \ \neg\Delta_{th}(\tau)
\end{array}}{\Delta;\Gamma;\Phi\vdash e.m(e_1,\ldots,e_n):\tau|\Phi'}$$
$$\text{(TE-Method)}$$

$$\frac{\begin{array}{c}
\Delta;\Gamma;\Phi\vdash e_1:C_1|\Phi_1\ \ldots\ \Delta;\Gamma;\Phi_{n-1}\vdash e_n:C_n|\Phi_n\\
\Delta_c(C)=C'_1,\ldots,C'_n,\Phi_C \quad C_1\prec C'_1\ \ldots\ C_n\prec C'_n\\
\mathit{cnbody}(C)=(x_1,\ldots,x_n,\mathbf{super}(e'_1,\ldots,e'_k);\bar{s})\\
\mathit{if}\ \ \Phi_C\uparrow\ \ \mathit{then}\\
\left(\begin{array}{c}
\mathit{if}\ \ \Delta;(C,C)\vdash\mathit{loop}(C,C)\\
(\mathit{then}\ \ \Delta_1=\Delta[\Delta_c(C)\mapsto C'_1,\ldots,C'_n,\emptyset]\ \ \mathit{else}\ \ \Delta_1=\Delta)\\
\Gamma_1 = x_1:C'_1,\ldots,x_n:C'_n,\mathbf{this}:C\\
\Delta_1;\Gamma_1;\emptyset\vdash\mathbf{super}(e'_1,\ldots,e'_k);:\mathbf{void}|\Phi_s \quad \Delta_1;\Gamma_1;\Phi_s\vdash\bar{s}:\mathbf{void}|\Phi''
\end{array}\right)\\
\mathit{else}\ \ \Phi''=\Phi_C\\
\mathit{if}\ \ \Delta_{th}(C)\ \ \mathit{then}\ \ (\Phi_n=\Phi\ \wedge\ \Phi''=\emptyset)\\
\Phi'_C=\mathit{replace}(\mathbf{this},x_1,\ldots,x_n,\Phi'',\mathbf{this}_C,e_1,\ldots,e_n)\wedge\Phi'_C\downarrow\\
\mathit{if}\ \ \Phi'_C\neq\emptyset\ \ \mathit{then}\ \ \mathit{check}(\Phi'_C,\Phi_n)\\
\Phi'=\Phi_n\cup\Phi'_C
\end{array}}{\Delta;\Gamma;\Phi\vdash\mathbf{new}\ C(e_1,\ldots,e_n):C|\Phi'}$$
$$\text{(TE-New)}$$

Figure 5 – Typing rules for some expressions. The symbol "↑" stands for undefined and "↓" for defined.



```
C y;
y = new C();
(C')y.m(); //Class C is a subclass of C'.
           //Method m is defined in C'.
           //Class C' has a field f1 of type C1
           //which has a field f of type Thread and
           //a field g which does not have any reference
           //to a Thread object.
           //The up-casting (C')y is ill-typed because y
           //has a reference to a Thread object.
```

```
C' x;
x = new C();
x.f1 = new C1();
x.f1.f = new Thread();
x.f1.g = y.f1.g;
x.m();
```

Figure 6 – Converting an ill-typed correct program to a well-typed program.

$(C')e$. The code also includes assignments that assign the fields of $e$ to those of $x$, i.e., "$x.f = e.f$;" for any field $f$ that appears in class $C'$. Suppose that the constructor of $C$ takes no parameter. It is worth mentioning that the operator **new** creates a new object. This prevents aliasing, and thus, the invocation of **start** on this object is safe. This conversion is also given, in the first row, in Table 1. A list of similar conversions can be found in this table.

The premise $\neg\Delta_{th}(C)$ may also be false because $\exists f, C''.\ (C'' = \Delta_f(C)(f) \wedge \Delta_{th}(C''))$ is true. This is the case for the ill-typed program given in Fig. 6 where field `f1` in class `C` is of a type—class—having a field `f` of type **Thread**. Here, the conversion stated above for the case of the subclasses of **Thread** is extended. In effect, the fields containing threads are not directly assigned, but we first create new objects as in the case above and then assign these new objects to the fields. This is performed for the fields `f1` and `f` in the well-typed program given in Fig. 6. Note that this program can be safely extended with both `y.f1.f.start();` and `x.f1.f.start();`, although `(C')y.f1.f.start();` cannot be used instead of the latter in the original code. It is worthy of mention that although some more instantiations may be made in the substitute code, they are required for correct invocations of **start** on non-null **Thread** objects. Finally, the case $\exists C''.\,(C'' \prec C \wedge \Delta_{th}(C''))$ is treated similarly.

### 3.3.2 Typing Method Invocations

The rule TE-Method is for type checking the invocation of a method $m$ on an expression $e$ with parameters $e_1, \ldots, e_n$. The type of $e$ is first derived according to $\Delta$, $\Gamma$, and $\Phi$. It includes $\Phi_0$ that is considered as part of the context in type checking the parameters to $m$. As stated earlier, $\Phi_{i-1}$ is contained within $\Phi_i$. Therefore, $\Phi_n$ consists of the elements of $\Phi$ together with those expressions on which the invocation of **start** is predicted as a result of type checking the expressions $e$ and $e_1, \ldots, e_n$. The function *giveMethod* assigns **run** or $m$ to $m_g$ as defined in Fig. 4. The types of the arguments of $m_g$ are obtained from $\Delta_m$. The types $C_1, \ldots, C_n$ are then checked to comply with these types.

**Preventing Multiple Start of Threads.** If $m$ is **start** and the class of $e$ is a subclass of **Thread**, $e$ should not be a member of $\Phi_n$—the function *CheckIfStart* in Fig. 4 is used for this purpose. If $m$ is **start**, $\Phi_{current}$ is set to $\{e\}$ and is included in $\Phi'$. The invocations of **start** in the body of $m_g$ which are collected in the set $\Phi_{m_g}$ should also be checked against $\Phi_n$ and $\Phi_{current}$, through the function *check*. However, $m_g$ may not be type checked yet, and therefore, $\Phi_{m_g}$ may still be undefined. If so, the body of $m_g$ in class $C_0$ is type checked against the typing context containing $\Gamma_1$ which includes the parameters to the method mapped to their types and variable **this** mapped to type $C_0$.

**Typing Recursive Methods.** The judgment $\Delta; (C_0, m_g) \vdash loop(C_0, m_g)$ states that method $m_g$ of class $C_0$ is a member of some loop—the rules defining such a judgment are given in the online appendix of this paper [ISF16a]. Note that if $m_g$ is in some loop of invocations, type checking its body may repeat infinitely many times. This is because type checking the body of $m_g$ requires $\Phi_{m_g}$ which is still undefined. To prevent this, typing context should be constructed in such a way that type checking remains decidable. In doing so, $\Phi_{m_g}$ is set to the empty set and the resulting class table $\Delta_1$ is considered in type checking the body. As it is assumed that invocations of **start** in recursive methods do not contain parameters, such invocations are collected in the set $\Phi''$ by one-pass checking of the body. Note that $\Delta_1$ is considered temporarily in type checking the body of $m_g$. The correct content of $\Phi_{m_g}$ is later assigned to this set by the rule T-MDefn in Fig. 9.

As an example, consider the following code. It is a loop in the form of a method which invokes itself.

```
void m() {
    Thread t;
    t = new Thread();
    t.start();
    this.m();
}
```

Assume that $m$ is invoked somewhere in the program. When this invocation is type checked through TE-Method, the body of $m$ is also type checked if $\Phi_m$ is undefined. In type checking the body of $m$, $\Phi_m$ is temporarily set to the empty set. Otherwise, the type checking of "`this.m()`" leads to another type checking of the body of $m$ which, in turn, makes the whole process endless. Note that the variable "`t`" is collected in the set $\Phi''$ derived as the set containing the expressions on which **start** is invoked in the body of $m$.

**Required Replacements.** Since $\Phi_n$ and $\Phi_{current}$ contain the suspicious expressions that are composed of actual parameters, an occurrence of a formal parameter in $\Phi_{m_g}$ is replaced by its corresponding actual parameter through the function *replace* in Fig. 4. For the same reason, $e$ is also substituted for the local variable **this**. If these replacements convert two different expressions in $\Phi_{m_g}$ to the same, type checking fails. This signifies probable double invocation of **start** on the same thread at run-time. For a successful replacement, it is also checked if $\Phi'_{m_g}$ is non-empty. If so, the contents of $\Phi'_{m_g}$ are examined by the function *check*. The set $\Phi'$ is then constructed as the union of $\Phi_n$, $\Phi_{current}$, and $\Phi'_{m_g}$.

**The Bad Aliasing Due to Method Invocations.** Note that the expression returned by a method is an alias for the invocation of that method on an object. The

$$\frac{\Delta;\Gamma;\Phi \vdash pe : \tau|\Phi'}{\Delta;\Gamma;\Phi \vdash pe; : \textbf{void}|\Phi'}$$
(TS-PE)

$$\frac{\begin{array}{c}\Delta;\Gamma;\Phi \vdash e_1 : C_1|\Phi_1 \quad \Delta;\Gamma;\Phi_1 \vdash e_2 : C_2|\Phi_2 \\ \Delta;\Gamma;\Phi_2 \vdash \bar{s}_1 : \textbf{void}|\Phi_3 \quad \Delta;\Gamma;\Phi_2 \vdash \bar{s}_2 : \textbf{void}|\Phi_4 \\ C_1 \prec C_2 \vee C_2 \prec C_1 \\ \Phi' = \Phi_3 \cup \Phi_4\end{array}}{\Delta;\Gamma;\Phi \vdash \textbf{if } (e_1 == e_2)\{\bar{s}_1\} \textbf{ else } \{\bar{s}_2\} : \textbf{void}|\Phi'}$$
(TS-If)

$$\frac{\begin{array}{c}\Delta;\Gamma;\Phi \vdash e' : C_1|\Phi' \quad \Delta;\Gamma;\Phi' \vdash e : C_2|\Phi'' \\ \Delta_f(C_1)(f) = C_3 \quad C_2 \prec C_3 \\ e = \textbf{null} \vee e = \textbf{new } C_2(e_1,\ldots,e_n) \vee \neg\Delta_{th}(C_2)\end{array}}{\Delta;\Gamma;\Phi \vdash e'.f = e; : \textbf{void}|\Phi''}$$
(TS-FieldWrite)

$$\frac{\begin{array}{c}\Delta;\Gamma;\Phi \vdash x : C_3|\Phi' \quad \Delta;\Gamma;\Phi' \vdash e : C_2|\Phi'' \\ C_2 \prec C_3 \quad x \neq \textbf{this} \\ e = \textbf{null} \vee e = \textbf{new } C_2(e_1,\ldots,e_n) \vee \neg\Delta_{th}(C_2)\end{array}}{\Delta;\Gamma;\Phi \vdash x = e; : \textbf{void}|\Phi''}$$
(TS-VarWrite)

$$\frac{\Delta;\Gamma, x : C;\Phi \vdash s_1 \ldots s_n : \tau|\Phi'}{\Delta;\Gamma;\Phi \vdash C\ x; s_1 \ldots s_n : \tau|\Phi'}$$
(TS-Intro)

$$\frac{\begin{array}{c}s_1 \neq C\ x; \\ \Delta;\Gamma;\Phi \vdash s_1 : \textbf{void}|\Phi_1 \quad \Delta;\Gamma;\Phi_1 \vdash s_2 \ldots s_n : \tau|\Phi'\end{array}}{\Delta;\Gamma;\Phi \vdash s_1 s_2 \ldots s_n : \tau|\Phi'}$$
(TS-Seq)

Figure 7 – Typing rules for some statements.

last premise prevents such an aliasing if the resulting aliases are threads or contain threads in their fields.

### 3.3.3 Typing Constructor Invocations

The rule TE-New type checks the invocation of class constructors. Since constructors are, in fact, a special kind of methods, most of the premises of TE-Method are adapted for this rule. Note that the variable **this** in $\Phi_n$ points to the current object. However, the same variable in the body of constructor points to the new object that is returned by its invocation. To distinguish between these objects, every occurrence of **this** in $\Phi_C$ is replaced by $\textbf{this}_C$. In this way, invocations of **start** on expressions containing **this** are not mistakenly prevented due to such invocations in constructor $C$. Note that variable **this** is not changed in the body being type checked or in the mapping $\Gamma$. Finally, if $\Delta_{th}(C)$ is true, we should have $\Phi_n = \Phi$ and $\Phi_C = \emptyset$. This guarantees that no invocation of **start** occurs in the parameters $e_1,\ldots,e_n$ or in the body of the constructor $C$.

## 3.4 Typing Rules for Statements

The typing rules of MTMJ for some statements are given in Fig. 7. The rules TS-PE and TS-INTRO simply repeat $\Phi'$ of their premises in their conclusion.

### 3.4.1 Typing Conditional Statements

The rule TS-IF is for type checking if-then-else statements. The expressions $e_1$ and $e_2$ in the condition part of the statement are first type checked. The set $\Phi_1$ obtained from type checking $e_1$ appears in the context for typing $e_2$. Similarly, the set $\Phi_2$ obtained from type checking $e_2$ is considered as part of typing context for the two branches $\bar{s}_1$ and $\bar{s}_2$. The set $\Phi'$ in the conclusion of the rule would then be the union of $\Phi_3$ and $\Phi_4$ resulted from type checking the two branches. In this way, any invocation of **start** in the branch which may not be executed at run-time is also considered. This is typical of any static approach to the prevention of errors in conditional structures. In fact, the knowledge of which branch actually occurs depends on run-time values taken by expressions. Although the value of some expressions, e.g., **null**, may be known a priori, this is not generally true. The following code is an example of a conditional structure.

```
if (x.m().f == y.g.n()){t1.start();}
                         else {t2.start();}
```

The expression "`x.m().f`" represents an access to the field "`f`" of the object pointed by the value returned by "`m`" in which $m$ itself is a method of the object identified by "`x`". As the value of such an expression may not be known statically, the type checker assumes that both branches may be executed at run-time. Therefore, both `t1` and `t2` are collected in $\Phi'$. It should be noted that in a valid if-then-else statement, the types of the two expressions compared in the condition part of the statement should be related by the subclass relation.

### 3.4.2 Typing Sequences of Statements

A sequence of statements $s_1, \ldots, s_n$ is type checked by the rule TS-SEQ. As seen, the set $\Phi_1$ obtained from type checking $s_1$ appears in the typing context for the tail sequence. This rule cannot be applied to those sequences beginning with a variable declaration—such sequences are type checked by the rule TS-INTRO.

### 3.4.3 Typing Assignments

The rules TS-FIELDWRITE and TS-VARWRITE are for type checking the assignment of an expression $e$ to a field $e'.f$ and to a variable $x$, respectively. According to these rules, the left side of the assignment is first type checked. The set $\Phi'$ then appears in the typing context for $e$. The set $\Phi''$ in the conclusion of the rules is the one resulted from type checking $e$. The type of $e$ should also be a subclass of the type of the left side of the assignment. Since the left side of an assignment is an alias for its right side, we should be certain that the right side is not—or does not contain—a thread. This is realized by adding $\neg\Delta_{th}(C_2)$ to the premises of the rule. Nevertheless, the aliasing stated above does not occur for an expression $e$ of the form **new** $C_2(e_1, \ldots, e_n)$, even if $\Delta_{th}(C_2)$ is true. This is because the new object identifier returned by **new** $C_2(e_1, \ldots, e_n)$ is assigned to the left side of the assignment such that the corresponding object can only be accessed through the expression in the left. It is worth noting that an assignment not satisfying the last line of premises in

$$\frac{C \prec_1 \textbf{Object}}{\Delta; \Gamma, \textbf{this} : C; \Phi \vdash \textbf{super}();: \textbf{void}|\Phi}$$
$$(\text{T-CO}\textsc{bject})$$

$$\frac{\begin{array}{c} \Gamma = \Gamma', \textbf{this} : C_0 \quad \Gamma' \neq *, \textbf{this} : C', * \quad C_0 \prec_1 C \\ \Delta; \Gamma'; \Phi \vdash e_1 : C_1|\Phi_1 \quad \ldots \quad \Delta; \Gamma'; \Phi_{n-1} \vdash e_n : C_n|\Phi_n \\ \Delta_c(C) = C'_1, \ldots, C'_n, \Phi_C \quad C_1 \prec C'_1 \quad \ldots \quad C_n \prec C'_n \\ \mathit{cnbody}(C) = (x_1, \ldots, x_n, \textbf{super}(e'_1, \ldots, e'_k); \bar{s}) \\ \mathit{if}\ \ \Phi_C \uparrow \ \ \mathit{then} \\ \left( \begin{array}{c} \mathit{if}\ \ \Delta; (C, C) \vdash \mathit{loop}(C, C) \\ (\mathit{then}\ \ \Delta_1 = \Delta[\Delta_c(C) \mapsto C'_1, \ldots, C'_n, \emptyset]\ \ \mathit{else}\ \ \Delta_1 = \Delta) \\ \Gamma_1 = x_1 : C'_1, \ldots, x_n : C'_n, \textbf{this} : C \\ \Delta_1; \Gamma_1; \emptyset \vdash \textbf{super}(e'_1, \ldots, e'_k);: \textbf{void}|\Phi_s \\ \Delta_1; \Gamma_1; \Phi_s \vdash \bar{s} : \textbf{void}|\Phi'' \end{array} \right) \\ \mathit{else}\ \ \Phi'' = \Phi_C \\ \Phi'_C = \mathit{replace}(x_1, \ldots, x_n, \Phi'', e_1, \ldots, e_n) \wedge \Phi'_C \downarrow \\ \mathit{if}\ \ \Phi'_C \neq \emptyset\ \ \mathit{then}\ \ \mathit{check}(\Phi'_C, \Phi_n) \\ \Phi' = \Phi_n \cup \Phi'_C \end{array}}{\Delta; \Gamma; \Phi \vdash \textbf{super}(e_1, \ldots, e_n);: \textbf{void}|\Phi'}$$
$$(\text{T-CS}\textsc{uper})$$

Figure 8 – Typing rules for the **super** call.

TS-FieldWrite or TS-VarWrite can be rewritten as a well-typed program with the same functionality.

## 3.5 Typing Rules for the **super** Call

Figure 8 shows typing rules for the **super** call. The rule T-CObject is applicable if **Object** is the direct superclass of the current class. T-CObject extends the corresponding rule of MJ by adding $\Phi$ to the hypotheses and the consequent of the typing judgment. In the rule T-CSuper, $C$ is the direct superclass of $C_0$. Since no valid object is created before executing the constructor of $C$, the variable **this** may not appear in the parameters $e_1, \ldots, e_n$. To enforce this, the typing context for these parameters, $\Gamma'$, is obtained by removing "**this** : $C_0$" from $\Gamma$. Most of the premises of T-CSuper are the same as TE-New.

## 3.6 Typing Rules for the Whole Program

The rules given in Fig. 9 type check a program. According to T-ProgDef, a program $P$ is well-typed if the constructors and methods of its classes are well-typed. The function $chTP$ in Fig. 4 also checks for the conditions that should be satisfied by overridden methods. Moreover, the main body of the program, $\bar{s}$, should be typable. Note that the context $\Gamma$ is empty because the main body has no argument. The variable **this** is not included in $\Gamma$ either because the main body cannot access such a variable.

The judgment $\Delta \vdash C\ cok|\Delta'$ defined by the rule T-CDefn states that the constructor of class $C$ is well-typed. Moreover, the type of the constructor is updated and a new mapping $\Delta'$ is produced. As seen in the premises of this rule, the body of the constructor of $C$ that is composed of a **super** call followed by a sequence of statements should be typable according to the typing rules of MTMJ. Note that if $C$ is in a loop of invocations, $\Phi_C$ is set to the empty set when its body is type checked. The typing context $\Gamma$ includes typed arguments as determined by $\Delta_c(C)$. Moreover,

$$\dfrac{\begin{array}{c}
\left(\begin{array}{c} \mathit{if}\ \ m = \mathbf{start}\ \ \mathit{then} \\ \left(\begin{array}{c} C \prec \mathbf{Thread} \wedge C \neq \mathbf{Thread} \wedge \\ P = *\ \mathbf{class}\ C\ \mathbf{extends}\ C'\{*\ \mathit{cnd}\ \mathit{md}_1 \dots \mathit{md}_n\}\ * \end{array}\right) \\ \Rightarrow \forall 1 \leq i \leq n.\ \mathit{md}_i \neq \mathbf{void}\ \mathbf{start}()\{*\} \end{array}\right) \\
m_g = \mathit{giveMethod}(C, m) \\
\Delta_m(C)(m_g) = C_1, \dots, C_n \dashv\langle * \rangle\rightarrowtail \tau \\
\mathit{if}\ \ \tau \neq \mathbf{void}\ \ \mathit{then}\ \ \neg\Delta_{th}(\tau) \\
\mathit{if}\ \ \Delta; (C, m_g) \vdash \mathit{loop}(C, m_g) \\
(\mathit{then}\ \ \Delta_1 = \Delta[\Delta_m(C)(m_g) \mapsto C_1, \dots, C_n \dashv\langle \emptyset \rangle\rightarrowtail \tau]\ \ \mathit{else}\ \ \Delta_1 = \Delta) \\
\mathit{mbody}(C, m_g) = (x_1, \dots, x_n, \bar{s}) \\
\Gamma = x_1 : C_1, \dots, x_n : C_n, \mathbf{this} : C \\
\Delta_1; \Gamma; \emptyset \vdash \bar{s} : \tau | \Phi' \\
\Delta' = \Delta[\Delta_m(C)(m_g) \mapsto C_1, \dots, C_n \dashv\langle \Phi' \rangle\rightarrowtail \tau]
\end{array}}{\Delta \vdash \mathit{mbody}(C, m)\ \mathit{ok} | \Delta'}$$
(T-MDefn)

$$\dfrac{\begin{array}{c}
\mathit{dom}(\Delta_m(C)) = \{m_1, \dots, m_n\} \quad \Delta_0 = \Delta \\
\Delta_0 \vdash \mathit{mbody}(C, m_1)\ \mathit{ok}|\Delta_1 \quad \dots \quad \Delta_{n-1} \vdash \mathit{mbody}(C, m_n)\ \mathit{ok}|\Delta_n
\end{array}}{\Delta \vdash C\ \mathit{mok}|\Delta_n}$$
(T-MBodies)

$$\dfrac{\begin{array}{c}
\Delta_c(C) = C_1, \dots, C_n, * \\
\mathit{if}\ \ \Delta; (C, C) \vdash \mathit{loop}(C, C) \\
(\mathit{then}\ \ \Delta_1 = \Delta[\Delta_c(C) \mapsto C_1, \dots, C_n, \emptyset]\ \ \mathit{else}\ \ \Delta_1 = \Delta) \\
\mathit{cnbody}(C) = (x_1, \dots, x_n, \mathbf{super}(e'_1, \dots, e'_k); \bar{s}) \\
\Gamma = x_1 : C_1, \dots, x_n : C_n, \mathbf{this} : C \\
\Delta_1; \Gamma; \emptyset \vdash \mathbf{super}(e'_1, \dots, e'_k);: \mathbf{void}|\Phi_s \\
\Delta_1; \Gamma; \Phi_s \vdash \bar{s} : \mathbf{void}|\Phi' \\
\mathit{if}\ \ \Delta_{th}(C)\ \ \mathit{then}\ \ \Phi' = \emptyset \\
\Delta' = \Delta[\Delta_c(C) \mapsto C_1, \dots, C_n, \Phi']
\end{array}}{\Delta \vdash C\ \mathit{cok}|\Delta'}$$
(T-CDefn)

$$\dfrac{\begin{array}{c}
\mathit{dom}(\Delta) = \{C_1, \dots, C_n\} \quad \Delta_0 = \Delta \\
\Delta_0 \vdash C_1\ \mathit{cok}|\Delta_1 \quad \dots \quad \Delta_{n-1} \vdash C_n\ \mathit{cok}|\Delta_n \\
\Delta_n \vdash C_1\ \mathit{mok}|\Delta_{n+1} \quad \dots \quad \Delta_{2n-1} \vdash C_n\ \mathit{mok}|\Delta_{2n} \\
\mathit{dom}(\Delta_m(C_i)) = \{m_1^{C_i}, \dots, m_{k_{C_i}}^{C_i}\} \ \ (i = 1, \dots, n) \\
\forall 1 \leq i \leq n,\ 1 \leq j \leq k_{C_i}.\, \mathit{chTP}(\Delta_{2n}, m_j^{C_i}, C_i) \\
P = *;\ \bar{s} \quad \Delta_{2n}; \emptyset; \emptyset \vdash \bar{s} : \mathbf{void}|\Phi
\end{array}}{\Delta \vdash P\ \mathit{ok}}$$
(T-ProgDef)

Figure 9 – Typing programs.

| | | | | | |
|---|---|---|---|---|---|
| $s_1$ | $::=$ | `Thread1 th;` | 1 | $::=$ | TS-Intro |
| $s_2$ | $::=$ | `th = new Thread1();` | 2 | $::=$ | TS-Seq |
| $s_3$ | $::=$ | `th.start();` | 3 | $::=$ | TS-VarWrite |
| $s_4$ | $::=$ | `(Thread)th.start();` | 4 | $::=$ | TS-PE |
| $e_1$ | $::=$ | `th.start()` | 5 | $::=$ | TE-Method |
| $e_2$ | $::=$ | `(Thread)th.start()` | 6 | $::=$ | TE-UpCast |
| $\Gamma'$ | $::=$ | $\Gamma, \texttt{th} : \texttt{Thread1}$ | | | |

$$
{}^{1}\dfrac{{}^{2}\dfrac{{}^{3}\dfrac{\checkmark}{\Delta;\Gamma';\emptyset \vdash s_2 : \textbf{void}|\emptyset} \qquad {}^{2}\dfrac{{}^{4}\dfrac{{}^{5}\dfrac{\checkmark}{\Delta;\Gamma';\emptyset \vdash e_1 : \textbf{void}|\{\text{th}\}}}{\Delta;\Gamma';\emptyset \vdash s_3 : \textbf{void}|\{\text{th}\}} \qquad {}^{4}\dfrac{{}^{5}\dfrac{{}^{6}\dfrac{\checkmark\ \Delta_{th}(\text{Thread1})}{\checkmark\ \Delta;\Gamma';\{\text{th}\} \vdash (\text{Thread})\text{th} :\uparrow}}{\Delta;\Gamma';\{\text{th}\} \vdash e_2 :\uparrow}}{\Delta;\Gamma';\{\text{th}\} \vdash s_4 :\uparrow}}{\Delta;\Gamma';\emptyset \vdash s_3 s_4 :\uparrow}}{\Delta;\Gamma';\emptyset \vdash s_2 s_3 s_4 :\uparrow}}{\Delta;\Gamma;\emptyset \vdash s_1 s_2 s_3 s_4 :\uparrow}
$$

Figure 10 – Typing Program A in Fig. 1.

the type of **this** is set to $C$. In this part of the type system, a class is type checked irrespective of the other parts of the program. Thus, the initial set $\Phi$ for type checking the **super** call is taken empty. The resulting set $\Phi_s$ is then used for type checking the rest of the body. The last premise requires that $\Phi_C$ should be empty if $\Delta_{th}(C)$ is true. This guarantees that the object created by invoking the constructor does not include any **start** invocation on threads. Although this may reject those class constructors that are not even invoked in the main body of the program, we add it to the premises of the rule to be consistent with the other rules of the type system.

The rule T-MBodies defines the judgment $\Delta \vdash C\ mok|\Delta_n$ which means that the methods of class $C$ are well-typed and $\Delta_n$ contains their types in which there is no undefined component type. Typable methods themselves are derived from the rule T-MDefn that defines the judgment $\Delta \vdash mbody(C, m)\ ok|\Delta'$. The first premise of T-MDefn avoids **start** from being overridden in the subclasses of **Thread**. To check the body that will actually execute, the function *giveMethod* assigns **run** to $m_g$ if $m$ is **start**. The return type of $m_g$ should match the one obtained from $\Delta_m(C)(m_g)$. It is worth noting that $\Delta_{th}$ should not be true for type $\tau$, as explained in TE-Method, to be consistent with type checking those methods invoked in the main body of the program. It has also been justified in our explanation of TE-Method that one-pass checking of the body of $m_g$ is sufficient to derive $\Phi_{m_g}$ even if this method is in some loop of invocations. Moreover, if $m_g$ is in some loop of invocations, checking its body against the new derived $\Phi_{m_g}$ is not required as no member of this set can be an alias for expressions on which **start** is invoked in the body.

## 3.7 Examples of Type Checking

Here, we give some examples to clarify how the typing rules of MTMJ can prevent double invocation of **start** on the same thread. In effect, it is shown that the programs in Fig. 1 are not typable in MTMJ.

In Program A, it is assumed that `Thread1` is a subclass of `Thread`. Therefore, $\Delta_{th}(\texttt{Thread1}) = true$. As shown in Fig. 10, the type checker can derive no type for this program. In this figure, the symbol "$\uparrow$" stands for "has not been derived yet", "✓" for the omitted premises that are satisfied, and "$::=$" for "is defined to be".

To type check "`(Thread)th.start()`" through the rule TE-Method, the expression "`(Thread)th`" should be type checked first. However, the rule TE-UpCast

| | | | | | |
|---|---|---|---|---|---|
| $s_1$ | ::= | `y.f = new Thread();` | 1 | ::= | TS-Seq |
| $s_2$ | ::= | `y.f.start();` | 2 | ::= | TS-FieldWrite |
| $s_3$ | ::= | `Cell x;` | 3 | ::= | TS-PE |
| $s_4$ | ::= | `x = y;` | 4 | ::= | TE-Method |
| $e_1$ | ::= | `y.f.start()` | 5 | ::= | TS-Intro |
| $\Gamma'$ | ::= | $\Gamma, y$ :`Cell` | 6 | ::= | TS-VarWrite |
| $\Gamma''$ | ::= | $\Gamma', x$ :`Cell` | | | |

$$
{}_{1}\dfrac{
{}_{2}\dfrac{\checkmark}{\Delta;\Gamma';\emptyset\vdash s_1 : \mathbf{void}|\emptyset}
\qquad
{}_{1}\dfrac{
{}_{3}\dfrac{{}_{4}\dfrac{\checkmark}{\Delta;\Gamma';\emptyset\vdash e_1 : \mathbf{void}|\{\text{y.f}\}}}{\Delta;\Gamma';\emptyset\vdash s_2 : \mathbf{void}|\{\text{y.f}\}}
\quad
{}_{5}\dfrac{{}_{6}\dfrac{\checkmark\ \Delta_{th}(\text{Cell})}{\Delta;\Gamma'';\{\text{y.f}\}\vdash s_4 :\uparrow}}{\Delta;\Gamma';\{\text{y.f}\}\vdash s_3 s_4 :\uparrow}
}{\Delta;\Gamma';\emptyset\vdash s_2 s_3 s_4 :\uparrow}
}{\Delta;\Gamma';\emptyset\vdash s_1 s_2 s_3 s_4 :\uparrow}
$$

**Figure 11** – Typing Program B in Fig. 1.

cannot be applied here, as it requires $\Delta_{th}$(`Thread1`) to be false. Note that when "`th.start()`" is type checked through TE-Method, "`th`" is added to the set $\Phi$. There is no other invocation of **start** in the method **run** of `Thread1`.

In Program B, the class `Cell` is supposed to have a field of type `Thread`, and thus, $\Delta_{th}$(`Cell`) $=$ *true*. Part of the type derivation illustrated in Fig. 11 shows that the program is ill-typed. In fact, the statement "`x = y;`" cannot be type checked through TS-VarWrite, because it requires the expression assigned to "`x`" to be either **null** or "`new Cell()`", or $\Delta_{th}$(`Cell`) be false. As seen, the expression "`y.f`" is added to $\Phi$ when "`y.f.start()`" is type checked.

## 3.8 Complexity of Type Checking

The statements in the main body of the program are type checked in order. Considering these statements as well as those that constitute the bodies of constructors and methods, the complexity of type checking can be computed. In fact, typing a program is composed of type checking the constructors and methods of that program. Assume that $S_m$ is the number of statements in the method **main**, $C$ is the number of classes in the program, $M$ is the maximum number of methods in the classes, $ST$ is the maximum number of statements in the methods and constructors, and $PA$ is the maximum number of parameters in the expressions appearing in the program. Here, by statements, we mean atomic statements having no statement within. For a conditional statement, for example, the statements appearing in its branches are also considered in calculating $ST$. Each statement may contain at most two expressions which themselves may contain several expressions as their parameters. Thus, for the methods and constructors that are not in some loop of invocations, the cost of type checking is in the order $\mathcal{O}(S_m + (C \times M \times ST \times PA))$. For such methods, it can be assumed that type checking proceeds in such a way that the types of all methods and constructors invoked in the body of a method have already been derived when that method is type checked. For a loop of invocations consisting of $n$ methods and constructors, type checking the bodies is performed in the order of $\mathcal{O}(n^2)$ steps. Therefore, the cost of type checking the methods and constructors making a loop of size $n$ is $\mathcal{O}(n^2 \times ST \times PA)$. It can be shown that cost of type checking a loop of $n$ methods and constructors can be reduced to the order of $\mathcal{O}(n)$ steps [ISF16b].

## 3.9 A Type Checker for MTMJ Programs

We have implemented the type system of MTMJ in the framework Xtext [xte13]. This framework takes the syntax of a language in an EBNF-like style and produces an ANTLR parser [Par07] for that language. There are several ways to encode a type system in Xtext. We have decided on Xsemantics [Bet11] which takes typing rules as comprising a set of premises and one conclusion. The complete set of codes developed as well as instructions for using the type checker of MTMJ can be found in [ISF16a] which also contains some example programs. Some of these programs are safe multithreaded applications that are successfully type checked by our type checker. This is not the case for ill-typed programs which do not comply with the requirements of the type system of MTMJ.

## 4 Dynamic Semantics

The operational semantics of MTMJ is given as an abstract machine comprising a set of states and the rules defining state transitions. As it is based on the semantics of MJ, we repeat shortly some parts of MJ's dynamic semantics here. In MJ, a state, also said a configuration, is a quadruple consisting of a heap, a variable stack, a closed frame, and a frame stack that are denoted by $H$, $VS$, $CF$, and $FS$, respectively. These components are defined as follows:

- $H$: A partial function that maps an object identifier to its class and field values.
- $VS$: A mapping from variable names to their types and values.
- $CF$: The next term to be evaluated.
- $FS$: The current evaluation context.

A value may be null or an object identifier. An object identifier itself is an abstraction of a low-level implementation, e.g., it may represent some kind of pointer. The variable stack is a list of method scopes mapping the variables of methods to their types and values—a method scope is denoted by $MS$. A method scope itself consists of several block scopes $BS$. The domain of a block scope is the variables appearing in a block within the corresponding method. A block may be the main body of the method or the body of a control structure. The operator "$\circ$" concatenates a $BS$ to an $MS$ or an $MS$ to $VS$—the evaluation of a variable begins from the innermost scope. The complete operational semantics of MJ can be found in [BPP03].

MTMJ extends the operational semantics of MJ with some new rules defining the behavior of concurrent threads. In the first step, a state is redefined as a global configuration made up of a heap $H$ and a function $TP$. The heap remains the same as in MJ. The function $TP$ takes a thread identifier and returns its local configuration. A local configuration of a thread $th$ is the triple $(VS_{th}, CF_{th}, FS_{th})$ of a variable stack, a closed frame, and a frame stack. For example, $CF_{th}$ is a term in the body of $th$.

Figure 12 elaborates on the operational semantics of MTMJ, where "$\twoheadrightarrow$" shows small-step transitions. The rule E-Translate states that if the term to be evaluated is not the invocation of **start**, the thread runs as an MJ program. By a nondeterministic choice, $th$ is the current thread and its body should be executed. The rule E-MethodVoidStart explains what happens when **start** is invoked on a thread. By

$$\frac{\begin{array}{c} th \in dom(TP_1) \quad TP_1(th) = (VS^1_{th}, CF^1_{th}, FS^1_{th}) \quad CF^1_{th} \neq th'.\mathbf{start}() \\ (H_1, VS^1_{th}, CF^1_{th}, FS^1_{th}) \to (H_2, VS^2_{th}, CF^2_{th}, FS^2_{th}) \text{ is an MJ transition} \\ TP_2 = TP_1[th \mapsto (VS^2_{th}, CF^2_{th}, FS^2_{th})] \end{array}}{(H_1, TP_1) \twoheadrightarrow (H_2, TP_2)}$$

(E-Translate)

$$\frac{\begin{array}{c} th' \in dom(TP_1) \quad TP_1(th') = (VS_{th'}, th.\mathbf{start}(), CF \circ FS'_{th'}) \\ H(th) = (C, \mathbb{F}) \quad C \prec \mathbf{Thread} \\ mbody(C, \mathbf{run}) = ((), \bar{s}_{th}) \quad th \notin dom(TP_1) \\ BS_{th} = \{\mathbf{this} \mapsto (th, C)\} \\ TP_2 = TP_1[th' \mapsto (VS_{th'}, CF, FS'_{th'})] \cup \{(th, ((BS_{th} \circ [\,]) \circ [\,], \bar{s}_{th}, (\mathbf{return}\ th;) \circ [\,]))\} \end{array}}{(H, TP_1) \twoheadrightarrow (H, TP_2)}$$

(E-MethodVoidStart)

$$\frac{\begin{array}{c} th \in dom(TP_1) \quad TP_1(th) = \mathbf{NPE} \vee TP_1(th) = \mathbf{CCE} \\ TP_2 = TP_1 \setminus \{(th, TP_1(th))\} \end{array}}{(H, TP_1) \twoheadrightarrow (H, TP_2)}$$

(E-DeleteThread)

Figure 12 – The operational semantics of MTMJ.

invoking **start** on an object $th$ initialized from **Thread** or its subclasses, $th$ is scheduled for execution—the current thread is $th'$. When $th'$ invokes $th.\mathbf{start}$, the thread identifier $th$ is added to the domain of $TP_2$ that maps $th$ to its initial local configuration. The closed frame $CF_{th}$ is the body of **run** in the class of $th$. The variable stack $VS_{th}$ is the concatenation of [ ]—which is the initial $VS$—and the method scope $MS = BS_{th} \circ [\,]$ of **run**. The term **return** $th$ is also pushed on top of the initial $FS$. By executing **return** $th$, the method scope of **run** is removed from the variable stack. $CF_{th'}$ is also set to $CF$ which means that the execution of $th'$ should be continued with the statements following the invocation of **start**.

According to the rule E-DeleteThread, a thread should be removed from $TP$ when it encounters run-time exceptions. Here, we only consider the two exceptions NullPointerException (**NPE**) and ClassCastException (**CCE**). The operational semantics concerning these exceptions has been given in MJ. For instance, $(H, VS, \mathbf{null}.f, FS)$ is a configuration that reduces to **NPE**. It should be noted that type safety can be provided even in the presence of run-time exceptions. Although we may prefer the type system to be able to trap all possible errors, it is not feasible, or even plausible, in real-life programming languages. If we insist on a completely static solution, we should then act in an overly conservative manner, and in turn, reject many healthy programs.

## 5 Type Soundness

We prove that MTMJ is strongly typed and ensures that **start** is invoked at most once on any **Thread** object at run-time. As run-time transitions of MTMJ are between configurations rather than expressions or statements, we should first define a well-typed configuration. In doing so, we extend the definition of a typable configuration $(H,\ VS,\ CF,\ FS)$ of MJ with respect to a class table $\Delta$ which is represented by the judgment $\Delta \vdash (H,\ VS,\ CF,\ FS) : \tau'$. This judgment holds if the evaluation process leads to a value of type $\tau'$ when it starts from the given configuration and follows the transitions of the operational semantics. It also holds if the evaluation process arrives at a valid error state. The following auxiliary judgments are useful in defining a typable configuration.

- $\Delta \vdash H\ ok$: The types of the field values of an object in $H$ are in compliance with the static types given by $\Delta_f(C)$, where $C$ is the type of that object.
- $\Delta, H \vdash VS\ ok$: The type of the value of any variable in $VS$ is a subclass of the static type of that variable.
- $\Delta, H, VS \vdash FS : \tau \to \tau'$: $FS$ is well-typed in the sense that given a value of type $\tau$, which is the result of evaluating the closed frame $CF$ of the configuration, its execution results in a value of type $\tau'$.

The above judgments are defined by a number of typing rules in MJ. For instance,

$$\frac{\Delta; \mathit{context}(H, VS) \vdash CF : \tau \quad \Delta, H, VS \vdash FS : \tau \to \tau'}{\Delta, H, VS \vdash CF \circ FS : \tau'' \to \tau'}$$

is one of the rules defining $\Delta, H, VS \vdash CF \circ FS : \tau \to \tau'$. In this rule, $CF$ is first type checked according to the typing rules of MJ for expressions and statements. The function *context* returns the corresponding typing context $\Gamma$ by extracting the types of object identifiers and variables as appeared in $H$ and $VS$. Here, if the head of $FS$ is an *open frame* containing a hole "$\bullet$" of type $\tau$, the hole will be replaced with the value obtained from evaluating $CF$. Thus, $FS$ should be of type $\tau \to \tau'$ being analogous to a function that takes a value of type $\tau$ and returns a value of type $\tau'$—there are a set of rules in MJ for type checking an $FS$ of the form $OF \circ FS'$ in which $OF$ is an open frame. If the head of $FS$ is a closed frame—containing no hole within—and the evaluation of $FS$ results in a value of type $\tau'$, its type is considered as $\tau \to \tau'$, where $\tau$ can be any type. From the premises, it can then be deduced that the return type of $CF \circ FS$ is $\tau'$. Again, as it has no hole in its head, the type of $CF \circ FS$ is taken $\tau'' \to \tau'$ for an arbitrary $\tau''$.

Prior to defining a well-typed global configuration in MTMJ, we first show that the following properties hold in MTMJ. The proof of lemmas and theorems of this section can be found through the web page containing the codes, software, and the other things developed as part of this research [ISF16a].

**Lemma 1.** *Any well-typed expression $e$ of type $C \prec$* **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. **new** *$C(\bar{e})$.*
4. **new** *$C'(\bar{e}).f_1.f_2.\cdots.f_n$.*

**Lemma 2.** *Every expression $e$ evaluating to a value in $dom(TP)$ is of the forms stated in Lemma 1.*

**Lemma 3.** *A well-typed expression $e$ of type $C \prec$* **Thread** *cannot be an alias for any well-typed $e'$ of type $C' \prec$* **Thread**.

**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 $th \in dom(TP)$.*

$$
\mathit{giveThreads}(H, VS, S) = \begin{cases} \emptyset & \mathit{if}\, VS = [\,] \\ \mathit{giveThreads}(H, VS', S) & \mathit{if}\, VS = [\,] \circ VS' \\ \mathit{giveTh}(H, BS, S) \cup & \mathit{if}\, VS = (BS \circ MS) \circ VS' \\ \mathit{giveThreads}(H, MS \circ VS', S) & \end{cases}
$$

$$
\mathit{giveTh}(H, BS, S) = \bigcup_{x \in dom(BS)} \begin{pmatrix} \{x | BS_V(x) \in S\} \cup \\ \mathit{fldTP}(\{x, BS_V(x)\}, BS_V(x), H, S) \end{pmatrix}
$$

$$
\mathit{fldTP}(E, v, H, S) = \begin{cases} \bigcup_{f \in dom(\mathbb{F})} \mathit{checkFld}(E, \mathbb{F}(f), H, S) & \mathit{if}\, v \neq \textbf{null}\, \mathit{and} \\ & \mathbb{F} = H_V(v) \\ \emptyset & \mathit{otherwise} \end{cases}
$$

$$
\mathit{checkFld}(E, v, H, S) = \begin{cases} \{e.f | e \in E\} \cup & \mathit{if}\, v \in S \\ \mathit{fldTP}(\{e.f | e \in E\} \cup \{v\}, v, H, S) & \\ \mathit{fldTP}(\{e.f | e \in E\} \cup \{v\}, v, H, S) & \mathit{otherwise} \end{cases}
$$

Figure 13 – The function *giveThreads* in which $BS_V(x)$ stands for the value of $x$ as returned by $BS$ and $H_V(v)$ is the field values of $v$ in $H$.

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

The rules defining a well-typed global configuration of MTMJ are given in Fig. 14. In the rules given in this figure, the function *giveThreads*, which is defined in Fig. 13, returns the set of those expressions in the body of a thread that are of the form (1) or (2) stated in Lemma 1 and their values are in $dom(TP)$. Note that a double invocation of **start** on the same thread does not occur for the other forms because, according to Lemma 4, such expressions return a thread identifier not started yet. For a thread $th$, *giveThreads* takes the heap $H$, the variable stack $VS$ associated with $th$, and $S$ that is equal to $dom(TP)$. Then, it partitions $VS$ into block scopes. For a block scope $BS$, *giveThreads* calls *giveTh* which returns the set of those local variables whose values are in $dom(TP)$. The function *giveTh* also investigates the field values of local variables through *fldTP* and *checkFld*. This continues recursively until **null** appears as a value. It should be noted that for an expression $x.f_1.\cdots.f_n$ returned by *giveThreads*, the function also returns expressions of the form $o_i.f_i.\cdots.f_n$, where $o_i$ is the value of $x.f_1.\cdots.f_{i-1}$.

According to the rule TG-CFG in Fig. 14, a global configuration $(H, TP)$ is of type $\tau$ if the following judgments are derivable.

1. $\Delta \vdash H\ ok$.
2. $\Delta, H \vdash TP : \tau$: From the rule TG-TP, $TP$ is of type $\tau$ if the following conditions are satisfied by every $th \in dom(TP)$.
    - The type of $th$ obtained from $H(th)$ is a subclass of **Thread**. In the premises of TG-TP, this is stated by $\Delta, H \vdash th :$ **Thread**.
    - $VS_{th}$ is well-typed in the context $\Delta, H$.

$$\frac{\Delta \vdash H\ ok \quad \Delta, H \vdash TP : \tau}{\Delta \vdash (H, TP) : \tau}$$
(TG-CFG)

$$\frac{\left(\begin{array}{c} \forall th \in dom(TP) \\ \Delta, H \vdash th : \textbf{Thread} \\ \Delta, H \vdash VS_{th}\ ok \\ \Phi_{th} = dom(TP) \cup giveThreads(H, VS_{th}, dom(TP)) \\ \Delta, H, VS_{th}, \Phi_{th} \vdash CF_{th} \circ FS_{th} : \textbf{void} \rightarrow \tau \end{array}\right)}{\Delta, H \vdash TP : \tau}$$
(TG-TP)

Figure 14 – A well-typed global configuration.

- The type system of MTMJ yields the type **void** $\rightarrow \tau$ for $CF_{th} \circ FS_{th}$. Therefore, the set of expressions on which **start** has already been invoked, i.e., $\Phi_{th}$, should appear as part of the typing context. Since a configuration is a run-time concept, in order to type check a configuration by the typing rules of MTMJ, one should build the context using the run-time information. In particular, $\Phi_{th}$ is considered as the set of all values on which **start** has been invoked together with those variables or fields that **start** has been invoked on their values—such values can be obtained from $VS$ and $H$. The former is $dom(TP)$ and the latter is obtained from $giveThreads(H, VS_{th}, dom(TP))$. It should be noted that our arguments for type soundness, given later as a theorem, is valid only if the set $\Phi_{th}$ does not include more so-called static expressions than those collected during type checking through the type system of MTMJ—by a static expression, we mean an expression containing no hole or value. That is, in the program text, **start** should be explicitly invoked on all static expressions collected in $\Phi_{th}$. This certainly happens because of what is stated by Corollary 1 and the fact that the value of an expression does not appear in $dom(TP)$ unless **start** is invoked on that value. Moreover, the pairs of actual/formal parameters excluded from Corollary 1 does not menace the validity of a prospective proof of type soundness, as the type system of MTMJ substitutes actual parameters for formal ones when checking for illegal invocation of **start**.

As seen in Fig. 14, MTMJ extends the judgment $\Delta, H, VS \vdash FS : \tau \rightarrow \tau'$ with $\Phi$ in its hypotheses. Thus, it also extends the rules defining such a judgment—these rules are given in the online appendix of this paper [ISF16a]. Some of these rules only add a $\Phi$ to the hypotheses of the judgments appearing in the corresponding rule of MJ. However, other rules check some new conditions.

To prove the soundness of the type system of MTMJ, we first define a terminal global configuration as comprising only threads whose execution is complete; that is, threads which have their frame stack empty and a value for their closed frame. This occurs only if the execution of **run** is complete.

**Definition 1.** *(Terminal Configuration). A global configuration* $(H, TP)$ *is said to be terminal if* $TP(th) = (VS_{th}, v, [\ ])$ *for every* $th \in dom(TP)$*, where* $v$ *is a value.*

In what follows, it is proved that MTMJ is type-safe, i.e., it satisfies the properties progress and preservation. That is, no well-typed nonterminal global configuration will get stuck and the types of well-typed configurations are preserved during semantic reductions.

**Theorem 1.** *(Progress). If $(H_1, TP_1) : \tau$, then either $(H_1, TP_1)$ is a terminal global configuration or $(H_1, TP_1) \twoheadrightarrow (H_2, TP_2)$ for some global configuration $(H_2, TP_2)$.*

**Theorem 2.** *(Type Preservation). If $(H_1, TP_1) : \tau$ and $(H_1, TP_1) \twoheadrightarrow (H_2, TP_2)$, then there exists a type $\tau'$ such that $(H_2, TP_2) : \tau'$ and $\tau' \prec \tau$.*

The type safety of MTMJ guarantees that no multiple invocation of **start** on the same thread happens for a typable program at run-time.

**Corollary 2.** *A well-typed program $P$ does not lead to multiple start of the same thread.*

*Proof.* As MTMJ is type-safe, no nonterminal global configuration gets stuck during the execution of $P$. Moreover, the dynamic semantics of MTMJ has no reduction for those global configurations involving multiple invocation of **start** on the same thread—see the rule E-MethodVoidStart. This completes the proof. □

## 6 Conclusion

In this paper, we have proposed MTMJ which is a multithreaded model language for Java. To do so, we have extended Middleweight Java with the primitives required for multithreading. The language is so designed that run-time misbehaviors can be predicted by the type system. In particular, MTMJ checks for erroneous multiple invocation of **start** statically, although it is dynamically checked in Java. To achieve this, MTMJ prohibits special kinds of aliasing through its type system. We have proved that MTMJ is type-safe, thereby trapping those programs whose execution results in multiple start of threads. Multiple invocation of an arbitrary method on the same object is not in general erroneous because, unlike **Thread** objects, such an object is not necessarily a handle for an executing entity. However, our approach is applicable to cases where one intends to prohibit multiple invocation of the same method on the same object. Extending MTMJ with other constructs, such as the synchronization construct, and modifying the type system of Java according to the typing rules of MTMJ deserve future research.

## References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 311–330, New York, NY, USA, 2002. ACM.

[And08] Android Open Source Project-Issue Tracker. Issue 972: IllegalThreadStateException in LunarLander (or SurfaceView?). `http://code.google.com/p/android/issues/detail?id=972`, 2008.

[AS15] Robert Atkey and Donald Sannella. ThreadSafe: Static analysis for Java concurrency. *Electronic Communications of the EASST*, 72:1–15, 2015.

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM.

[AY07] Alexander Ahern and Nobuko Yoshida. Formalising Java RMI with explicit code mobility. *Theoretical Computer Science*, 389(3):341–410, December 2007.

[BDSS10] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. A prototypical Java-like language with records and traits. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 129–138, New York, NY, USA, 2010. ACM.

[Bet11] Lorenzo Bettini. A DSL for writing type systems for xtext languages. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 31–40, New York, NY, USA, 2011. ACM.

[Boy01] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, may 2001.

[Boy04] Chandrasekhar Boyapati. *SafeJava: A unified type system for safe programming.* PhD thesis, Massachusetts Institute of Technology, 2004.

[BPP03] Gavin M. Bierman, Matthew J. Parkinson, and Andrew M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, 2003.

[BPSM99] Viviana Bono, Amit Patel, Vitaly Shmatikov, and John Mitchell. A core calculus of classes and objects. *Electronic Notes in Theoretical Computer Science*, 20:28–49, December 1999.

[Bug12a] Bugzilla@Mozilla. Bug 721924. `https://bugzilla.mozilla.org/show_bug.cgi?id=721924`, 2012.

[Bug12b] Bugzilla@Mozilla. Bug 722851. `https://bugzilla.mozilla.org/show_bug.cgi?id=722851`, 2012.

[BV99] Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 82–96, Denver, Colorado, USA, 1999. ACM.

[CGS$^+$05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek

Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[CÖSW13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[DCMYD06] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 328–352, Berlin, Heidelberg, 2006. Springer-Verlag.

[DDE$^+$11] Werner Michael Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 681–690, New York, NY, USA, 2011. ACM.

[DEK99] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, January 1999.

[DF04] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, ECOOP '04, pages 465–490. Springer Verlag, 2004.

[Die09] Werner Michael Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2009.

[DP09] Frédéric Dabrowski and David Pichardie. A certified data race analysis for a Java-like language. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 212–227, Berlin, Heidelberg, 2009. Springer-Verlag.

[DVE00] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited. Technical report, Imperial College London, 2000.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[FFLQ08] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Transactions on Programmming Languages and Systems*, 30(4):1–53, August 2008.

[FKF99] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical report, Rice University, 1999.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[GH99] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: reduction and typing. Technical report, University of Cambridge, Computer Laboratory, 1999.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Prentice Hall, 2005.

[Gro06] Christian Grothoff. *Expressive Type Systems for Object-Oriented Languages.* PhD thesis, University of California, Los Angeles, 2006.

[HAT$^+$04] Hesham H. Hallal, El-Hachemi Alikacem, William P. Tunney, Sergiy Boroday, and Alexandre Petrenko. Antipattern-based detection of deficiencies in Java multithreaded software. In *Proceedings of the Fourth International Conference on Quality Software*, QSIC '04, pages 258–267, Washington, DC, USA, 2004. IEEE Computer Society.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM.

[HSP05] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 13–19, New York, NY, USA, 2005. ACM.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[ISF16a] Zeinab Iranmanesh and Mehran S. Fallah. Multithreaded Middleweight Java: A Multithreaded Subset of Java. `http://ceit.aut.ac.ir/formalsecurity/mtmj`, 2016.

[ISF16b] Zeinab Iranmanesh and Mehran S. Fallah. Specification and static enforcement of scheduler-independent noninterference in a middleweight Java. *Computer Languages, Systems and Structures*, 46:20–43, November 2016.

[KA08] Matthew Kehrt and Jonathan Aldrich. A theory of linear objects. In *Proceedins of the Workshop on Foundations of Object Oriented Languages*, FOOL '08, San Francisco, CA, USA, 2008. ACM.

[Mic01] Microsoft Corporation. *C♯ language specifications.* Microsoft Press, 2001.

[Min96] Naftaly H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, ECCOP'96, pages 189–209, London, UK, 1996. Springer-Verlag.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[MPH99] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Fernuniversität Hagen, 1999.

[Nau96] Patrick Naughton. *The Java Handbook*. McGraw-Hill, California, 1996.

[NSPG08] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 457–474, New York, NY, USA, 2008. ACM.

[Ora02] Oracle Corporation and its affiliates. 1.4 doesn't throw IllegalThreadStateException. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4773384`, 2002.

[Ora05] Oracle Corporation and its affiliates. IllegalThreadStateException in start() when isAlive() returned false. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6287297`, 2005.

[Ora11] Oracle Corporation and its affiliates. Java platform, standard edition 7, API specification. `http://docs.oracle.com/javase/7/docs/api/`, 2011.

[OW10] Johan Östlund and Tobias Wrigstad. Welterweight Java. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 97–116, Berlin, Heidelberg, 2010. Springer-Verlag.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007.

[Pea10] David J. Pearce. A featherweight calculus for flow-sensitive type systems in Java. Technical report, Victoria University of Wellington, 2010.

[Pyt15] Python Software Foundation. *Python 3.4.3 Documentation, The Python Standard Library.* 2015.

[Saw98] Sushant Sawant. *Applying Static Analysis for Detecting Null Pointers in Java Programs.* DePaul University, 1998.

[SAWS05] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 83–94, New York, NY, USA, 2005. ACM.

[Str10] Rok Strniša. *Formalising, improving, and reusing the Java Module System.* PhD thesis, University of Cambridge, 2010.

[The15] The University of Maryland. FindBugs$^{tm}$ - Find bugs in Java programs. `http://findbugs.sourceforge.net/`, 2015.

[Wad90] Philip Wadler. Linear types can change the world! In *IFIP Working Conference on Programming Concepts and Methods.* North-Holland, 1990.

[WCG11] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Data-race exceptions have benefits beyond the memory model. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 30–36, New York, NY, USA, 2011. ACM.

[xte13] Xtext. `http://www.eclipse.org/Xtext/`, 2013.

## About the authors

**Zeinab Iranmanesh** received the BS and MS degrees in computer engineering from Sharif University of Technology. She received the PhD in computer engineering from Amirkabir University of Technology (Tehran Polytechnic) in 2016. Her research interests include programming languages and information security. Contact her at `ziranmanesh@aut.ac.ir`.

**Mehran S. Fallah** is an associate professor of computer engineering at Amirkabir University of Technology (Tehran Polytechnic). His research interests include programming languages, verification of computer systems, and language-based security. Contact him at `msfallah@aut.ac.ir`.