A Sound and Scalable Method for Static Data Race Detection in Java Programs
(Online Appendix)

Ali Ghanbari and Mehran S. Fallah

January 24, 2016

```
 1: | class Node extends Object {
 2: |     Object obj;
 3: |     Node next;
 4: |
 5: |     Node (Object obj, Node next) {
 6: |         this.obj = obj;
 7: |         this.next = next
 8: |     }
 9: | }
10: |
11: | class LinkedList extends Object {
12: |     Node first;
13: |
14: |     LinkedList () {
15: |         this.first = null
16: |     }
17: |
18: |     Node add (Object obj) {
19: |         this.first =
20: |             [new Node(obj, this.first)]$^{\ell_1}$
21: |     }
22: | }
23: | LinkedList l1 =
24: |     [new LinkedList()]$^{\ell_2}$;
25: |
26: | for (int i = 0; i < 25; i ++)
27: |     [l1.add([new Integer(i)]$^{\ell_3}$)]$^{\ell_4}$;
28: |
29: | LinkedList l2 =
30: |     [new LinkedList()]$^{\ell_5}$;
31: |
32: | [l2.add([new Float(3.14)]$^{\ell_6}$)]$^{\ell_7}$;
33: | [l2.add([new Float(2.72)]$^{\ell_8}$)]$^{\ell_9}$
```

Figure 1: An example program

A Points-to Analysis

Points-to and heap-shape analysis design techniques presented in [1, 2] form our ideas in the design of a precise, modular, and sound points-to analysis. The analysis is also able to determine multiplicity of objects that an expression may be evaluated to. Since our points-to analysis is context-sensitive, we expect that the information obtained from the analysis will be precise. In order to further increase the precision of the analysis, when analyzing methods in isolation, we have devised some annotations at method declaration points.

In the Java language that we are using here, every expression evaluates to an object, i.e. every expression is a kind of pointer. The goal of points-to analysis is to define a function, here we call it *mayPointsTo*, that maps each expression, which is associated with a unique label, to a value that somehow represents the target of the pointer. Such a function is obtainable from a points-to graph. In this section, we are going to present some important remarks regarding the design of a data-flow analysis to compute points-to graphs, through an example.

We are going to introduce three kinds of nodes in a points-to graph, namely, *internal*, *external*, and *return* nodes. An internal node represents a class of objects created within the analysis scope. We represent an internal node with i(id, m), where id is the unique identifier of the node, and m denotes the multiplicity of the node. An identifier for an internal node, similar to that of an abstract event, is a string of labels; such a choice for the identifier enables us in creating a context-sensitive analysis. An external node represents a parameter of the method being analyzed. Here, external nodes are denoted by e(n), where n is a natural number served as an identifier. External nodes are to be instantiated at call sites, i.e. each node is replaced by suitable internal node(s). The use of external nodes enables us to create a modular analysis. A return node is in fact an internal (or external) node that is marked as "return node," because the node represents an object that is returned by the method being analyzed.

We use an *abstract value* as a compile-time abstraction of the object an expression evaluates to. An abstract value is in fact a subset of the set of nodes of the points-to graph. The function *mayPointsTo* maps a label to an abstract value. In general terms, a points-to graph is a set of nodes and references between nodes (labelled edges, where labels are field names). Instead of delving into the, not very necessary, details of the analysis, we are going to explain how it does work through an example.

Figure 1, depicts a program fragment implementing a linked-list, and a client program using it. For ease of presentation, we have not followed pure syntax of ConcurrentJava$^+$. In particular, we have used constructor methods, for-loops, numerical types and constants, and have avoided the use of nested let construct to simulate sequencing of instructions. Additionally, we have labelled a number of expressions in the program by enclosing the desired expression inside brackets and writing its label as a superscript. The client code (lines 23-33) instantiates two linked lists l1, l2, and adds 25 elements to the first list, and two elements to the second.

Figure 2, illustrates points-to graph for each method and entry point expression of the program. For ease of presentation, we have named external nodes after the parameters they denote; other external nodes have received their official names. We are going to explain details of parts (3), (4) of the figure, other graphs are quite self-explanatory. In part (3), the edge from i(ℓ_1, ?) to the external node e(2) and the one from this to e(2) are due to the load instruction this.first at line 19 of the code. Since at that point the field first of the parameter this is not set, the analysis records a requirement on the external node this to have a reference to an object through that field. The requirement is depicted using an external reference. So, once the graph (which is a method summary for the method add) is to be instantiated, the requirement will be propagated if this is replaced by an external node. It will be satisfied if this is replaced by an internal node which have a reference

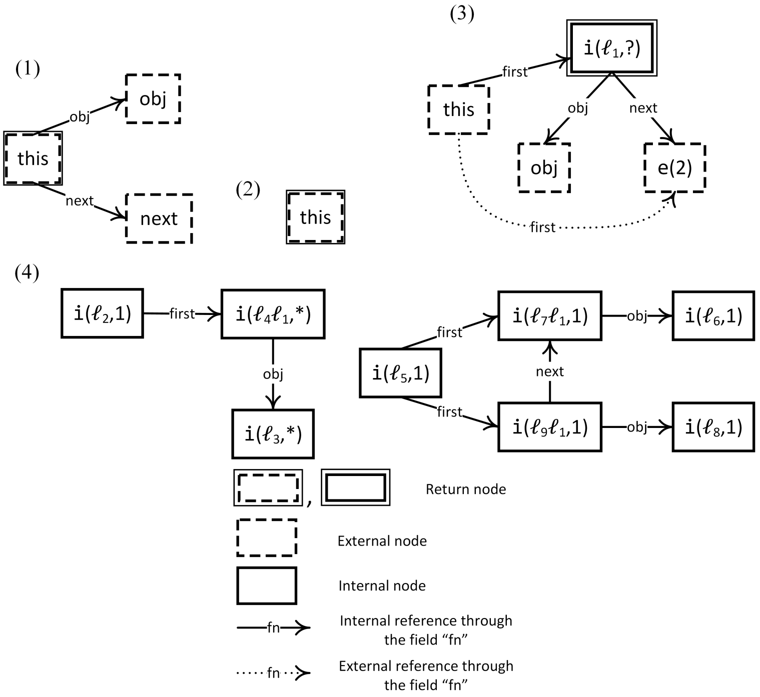


Figure 2: Points-to graphs for each fragment of the program of Figure 1. Parts (1), (2), and (3) illustrate points-to graphs (method summary) for final labels in bodies of constructor methods for the classes `Node` and `LinkedList`, and method `add`, respectively. Part (4) depicts points-to graph for final label of starting point expression of the program, i.e. lines 23-33 of the code.

to an internal or external node through its `first` field and, consequently, e(2) is replaced with an internal or external node. The requirement will be neglected if `this` is replaced by an internal node which does not have a reference through its `first` field.

Regarding part (4) of Figure 2, since the starting point expression will be executed by a single thread and only for one time, multiplicity of objects created at lines 24, 30, 32, and 33 will be 1. We have a `for`-loop at lines 26, 27; since (at compile-time) we are unaware of the number of the times the loop will iterate, we make a conservative assumption that the loop will iterate more than once, so the multiplicity for the objects created at line 27 are assumed to be *—i.e. each abstract object represents at least two run-time objects. First call to the method `add` appears at line 32. At that point `this` is replaced by the abstract object pointed to by `l1`, namely $\mathtt{i}(\ell_2, 1)$. Since the object does not have a reference through its field `first`, the requirement on the existence of e(2), and the edges incident with it, are neglected. The same thing is repeated at line 33 when `l2` does not have any element, i.e. it has no reference through its field `first`.

A.1 Alias-set Annotations

Perhaps our most important improvement over Whaley and Rinard's points-to analysis [1], is the introduction of the notion of alias sets, a mechanism to gain knowledge about calling contexts of a method. When we are analyzing methods in isolation, we have no prior knowledge about aliasing relations among parameters of a method. This lack of knowledge deteriorates the quality of information delivered by the analysis. On the other hand, a programmer writing the body of a method, according to the requirements on calling of the method, has a priori knowledge about aliasing relations of parameters of compatible types. By using alias-set annotations, which appears only at method headers, the programmer can group those parameters that *may alias* each other, in one alias set. Through the grouping, in fact, the programmer is declaring that the parameters, due to possible arguments provided at call sites, may point to each other. Figure 3 depicts an extended syntax for ConcurrentJava$^+$ that allows alias-set declarations.

$meth$	$::=$	$asdec^? \; t \; mn(par^*) \; asa^? \{e\}$	Method delaration
$asdec$	$::=$	`aset` $asid^*$	Alias-set identifier(s) declaration
par	$::=$	$t \; var \; asa^?$	Parameter declaration
asa	$::=$	`in` $asid$	Alias-set identifier assignment
$asid$	$\in$	$Identifiers$	Alias-set identifier

Figure 3: Extended syntactic categories

Each parameter, by default, is placed in a unique alias set. The annotation `aset`, at the beginning of a method declaration, is used to declare a finite set of distinct alias-set identifiers. The scope of each alias-set identifier is the method header. The annotation `in` n, where n is an alias set identifier, overrides the default alias set of a parameter, by placing it in the set n. The annotation for overriding the alias-set associated with the parameter `this`, appears after parameter list and before beginning of method body. In order to have a sound analysis, the program under analysis, at call-sites, must obey *aliasing policy* stated at method declaration points. We are going to enforce this requirement through a type system. The type system given here is, in fact, an extension to the standard type system presented in Section C.

Extending the type-rule WF Method

According to Figure 3, a method declaration is of the form shown below.

$$asdec\ t\ mn(t_1\ var_1\ asa_1, \ldots, t_k\ var_k\ asa_k)\ asa_0\{e\}$$

Since *asdec* could be the null-string, we declare our conditioned requirements on well-formedness of method declarations as follows. Note that a centered dot denotes string concatenation operator.

$$\begin{array}{l} asdec = \Lambda \implies asa_0 \cdot asa_1 \cdot \cdots \cdot asa_k = \Lambda \\ asdec = \texttt{aset}\ asid^* \implies \\ \quad (\textrm{ASIDO}\textsc{nce}(asid^*)\ \wedge\ \textrm{WFA}\textsc{ssignments}(asid^*, asa_0, asa_1, \ldots, asa_k)) \end{array}$$

Where the predicate ASIDOnce assures that there is no duplicate alias-set identifiers, and the predicate WFAssignments is used to check that each identifier is used at least once and at each assignment an already defined identifier has been used. Precise definitions for the predicates are given in the next section.

Extending the type-rule Exp Meth Call

According to the standard syntax of the subject language, a method call expression is of the following form.

$$e_0.mn(e_1, \ldots, e_k)$$

Extended inference rule, first, according to the type of e_0, identifies a family of classes from which the method mn will be applied on e_0, $e_1, \ldots, e_k$. Fortunately, the interface declared for each version of the method is the same. Thus, from the point of view of the rule, there is no difference between those versions. Assume the method mn is declared in the class introducing type t' as follows.

$$\begin{array}{l} e_0 : t' \\ asdec\ t\ mn(t_1\ var_1\ asa_1, \ldots, t_k\ var_k\ asa_k)\ asa_0\{e\} \in t' \end{array}$$

Where for each $0 \le i \le k$ we have $asa_i = (\texttt{in}\ a_i)^?$.

In order to extract alias-set identifiers from asa_i's, we define the auxiliary function *asid* as follows.

$$asid(s) = \left\{ \begin{array}{ll} a & ; s = \texttt{in}\ a \\ b & ; s = \Lambda \end{array} \right.$$

Where b is a fresh alias-set identifier.

Having the function *asid*, we define the following set that consists of pairs relating each alias-set identifier, associated with a parameter, with the set of abstract objects that an argument, corresponding to the parameter, may point to.

$$A = \{(asid(asa_i), mayPointsTo(e_i)) \mid 0 \le i \le k\}.$$

We also define the following auxiliary function:

$$f(A, a) = \bigcup \{P \mid (a, P) \in A\}$$

Finally, we express our requirement on method call sites as follows.

$$\forall a_1, a_2 \in dom(A).\ a_1 \ne a_2 \implies f(A, a_1) \cap f(A, a_2) = \emptyset.$$

The requirement states that two sets of abstract locations, pointed to by two arguments, corresponding to two parameters in two distinct alias sets, must be disjoint.

B The predicates ASIDOnce and WFAssignments

Precise definitions for the predicates are given in Figure 4 and Figure 5.

```
ASIDONCE($s$) $\triangleq$
$S := \emptyset$;
do
    if $a \in S$ then return false;
    else $S := S \cup \{a\}$;
    $a = tokenize(s, \text{“,”})$;
while $a \neq \Lambda$;
return true;
```

Figure 4: The predicate ASIDONCE

```
WFASSIGNMENTS($s, q_0, q_1, \ldots, q_k$) $\triangleq$
let $a_1 \cdot \text{“,”} \cdot \;\ldots\; \cdot \text{“,”} \cdot a_n = s$ in
    $S := \{a_1, \ldots, a_n\} \times \{0\}$;
for each $q \in \{q_0, \ldots, q_k\}$ do
    let ($q = \texttt{in}\ a$) in
        if $a \in dom(S)$ then $S(a) + +$;
        else return false;
for each $a \in dom(S)$ do
    if $S(a) < 1$ then return false;
return true;
```

Figure 5: The predicate WFASSIGNMENTS

$$\text{(WF METHOD)} \quad \frac{\begin{array}{c} P; E \vdash t \\ P; E, var_1 : t_1, \ldots, var_k : t_k \vdash e : t' \\ P; E \vdash t' <: t \end{array}}{P; E \vdash t\ md(t_1\ var_1, \ldots, t_k\ var_k)\{e\}}$$

$$\text{(EXP METH CALL)} \quad \frac{\begin{array}{c} P; E \vdash e_0 : t_0 \\ P; E \vdash (t\ md(t_{1..k}\ var_{1..k})\{e\}) \in t_0 \\ P; E \vdash e_i : t'_i \qquad P; E \vdash t'_i <: t_i \\ i \in 1..k \end{array}}{P; E \vdash e_0.md(e_{1..k}) : t}$$

Figure 6: An excerpt of standard type-system for CONCURRENTJAVA$^+$

C Static Semantics

Here, we are not going to present the whole set of rules of inference that form standard type-system for CONCURRENTJAVA$^+$—the type-system is presented in [3]. Instead, only those rules that are discussed in this paper are given. The rules are depicted in Figure 6, where the judgment $P; E \vdash t$ means that t is a well-formed type-name in the program P and the typing environment E. The judgment $P; E \vdash e : t$ declares that, in the program P and typing environment E, the expression e is of type t. The judgment $P; E \vdash t_1 <: t_2$ means that, in the program P and typing environment E, the type t_1 is a subtype of t_2. Finally,the judgment $P; E \vdash meth \in t$ states that, in the program P and typing environment E, the method *meth* is a declared or inherited method in a class which introduces the typename t.

D Operational Semantics

Following [4], in order to support a substitution-based operational semantics, we need to extend standard syntax of the language, given in Appendix C, in the following way, so that it includes addresses and the construct `insync` p `in` e.

$$e ::= \cdots \mid p \mid \texttt{insync}\ p\ \texttt{in}\ e$$

An address, or identifier, of an object is represented by p. The construct `insync` p `in` e indicates that the lock associated with the object p is held (by the evaluating thread) and the expression e is being evaluated. Note that these two newly added constructs should not appear in the source code.

Figure 8, illustrates operational semantics for CONCURRENTJAVA$^+$ by using an abstract machine. The machine, evaluates a programs through a sequence of states. A state is represented by a pair, in which the first component is an object store (i.e. a member of *Store*), and the second component is a sequence of thread identifier-expression pairs. A thread identifier-expression pair, as its name suggests, is a pair with its first component being a thread identifer, and second component being an expression (from extended syntax) that is to be evaluated. Since during program evaluation multiplicity of threads and objects are 1, we usually omit multiplicity information of identifiers. Roughly speaking, a thread identifier-expression pair denotes a thread of the program being evaluated. Value of a thread is the value computed for the expression associated with the thread. The result of a program is the value of the thread `main`, which appears as the first element of the sequence of threads located at the second component of a state. Newly spawned threads are appended to the end of the sequence of threads in a state. Let T_1, T_2 be sequence of threads, then $T_1 \cdot T_2$ denotes concatenation of the two sequences.

In a state, objects are kept inside an object store $\sigma \in \mathit{Store}$. An object store is a mapping from addresses to objects. An object is of the form $(\!|\mathit{db}|\!)_c^m$ with three components: (1) *db* is mapping from field identifiers to values (addresses or `null`), the mapping may be defined to be a sequence of the form $fn_1 = v_1, \ldots, fn_2 = v_n$, (2) m denotes state of the lock associated with the object, and (3) c is the class that the object belongs to. As before, Λ is used to represent an empty sequence, which in case of a field identifier-value mapping of an object, it means that the object does not have any declared or inherited fields.

Our abstract machine, in each transition, chooses a thread in a nondeterministic way in order to evaluate the expression associated with the thread. Obviously, such a nondeterministic choice simulates an arbitrary scheduler that interleaves the execution of different threads of a program. Since the value computed for a program may differ depending upon scheduling policy, we have defined *eval* to be a relation.

Evaluation of a program starts with an state containing an empty object store and a single thread, identifier of which is `main`. The expression associated with that thread is the initial expression of the program under evaluation. The evaluation proceeds according to state transition rules, and it halts once expressions associated with each thread evaluates to values (addresses or `null`). Hereafter, we shall use $\sigma\,[o/p]$ to denote an object store that agrees with σ, except in case of p, and the object associated with p is o. Furthermore, we will use $\sigma\,[v/p.fn]$ to denote an object store that agrees with σ, except in case of p, and the object associated with p is $\sigma(p)$, where its fn field is mapped to v.

Since state transition rules are straightforward, and compehensive explanations may be found in [4, 3], we will not explain the rules. The rule [Red New] uses auxiliary judgment $P; c \vdash_{init} \mathit{db}$ to determine the initial field map *db* for a newly created object. The other judgments are defined in the standard type system of the language given in Appendix C. The axiom and the rule of inference given in Figure 7, inductively prove the judgment $P; c \vdash_{init} \mathit{db}$.

$$(1)\ \frac{}{P; \texttt{Object} \vdash_{init} \Lambda}, \qquad (2)\ \frac{\begin{array}{c} P; \emptyset \vdash \texttt{class}\ c\ \texttt{extends}\ c'\{\mathit{field}_{1\ldots m}\ \mathit{meth}_{1\ldots n}\} \\ \mathit{field}_i = t_i\ fn_i \qquad i \in 1\ldots m \\ P; c' \vdash_{init} \mathit{db} \end{array}}{P; c \vdash_{init} \mathit{db}, fn_1 = \texttt{null}, \ldots, fn_m = \texttt{null}}$$

Figure 7: Inductive definition of $P; c \vdash_{init} \mathit{db}$

Now, through a series of definitions we are going to define a data race in terms of operational semantics given here. Let $e = \mathcal{E}\,[\texttt{insync}\ p\ \texttt{in}\ e']$ be an expression, where $\mathcal{E}$ is an evaluation context and e' is an expression. In such a case, we say that e is in *critical section* protected by p. Now, let $e = \mathcal{E}\,[p.fn]$ or $e = \mathcal{E}\,[p.fn = v]$ be an expression associated with a thread τ, then we say that the thread *accesses* field fn of the object p (abbreviated by $p.fn$). In the first case the access is of kind *reading* and in the second case the access would be a *writing*. A state contains *conflicting accesses* at $p.fn$ if the sequence of threads associated with the state has two threads accessing $p.fn$ where at least one of the accesses is of kind writing. A program has a *data race* if its evaluation reaches a state containing conflicting accesses. More precisely, the program $P = \mathit{defn}^*\ e$ has a data race, if $P \vdash <\emptyset, (\texttt{main}, e)> \hookrightarrow^* S$ where S contains a conflicting access—i.e. a data race.

E Soundness

In this section, we are going to prove the soundness of the non-modular concurrency analysis. According to Theorem 1 in Section 4, results obtained through modular and non-modular versions of the concurrency analysis are equivalent from a data race detection point of view. Thus, proving the soundness of non-modular version of the analysis, which is much simpler, implies the soundness of the modular analysis.

We shall conclude soundness of our non-modular analysis from the fact that if the abstract execution obtained for the final label of "main method" of a program (note that, as before, we suppose that initial expression of a program is placed in a method of a class) is *safe*, the program is free of data races. We assume that the program under analysis is free of any defects, e.g. `null`-pointer dereference, invalid type-casts, etc, other than data races. Furthermore, we suppose the following assumptions, conventions, and definitions.

- All (class) fields in the program under analysis are annotated with `norace` qualifier.

Program Evaluation:

$$eval(P,v) \iff P \vdash \langle \emptyset, (\mathtt{main}, e)\rangle \hookrightarrow^* \langle \sigma, (\mathtt{main}, v_1), \ldots, (\tau_n, v_n)\rangle\ ,$$
$$\text{where } P = defn^*\ e$$

State Space:

$$\begin{aligned}
&S \in State = Store \times ThreadSeq,\\
&\sigma \in Store = \{s \mid s : Addresses \to Objects\},\\
&T \in ThreadSeq = (\tau, e)^*,\\
&\tau \in TID,\\
&o \in Objects = \{(\!|db|\!)_c^m \mid db \in (fn_i = v_i)^* \wedge m \in \{\mathsf{locked}, \mathsf{unlocked}\}\},\\
&p \in Addresses = \{p_0, p_1, \ldots\}.
\end{aligned}$$

Evaluation Contexts:

$$\begin{aligned}
\mathcal{E} ::= {} & [] \mid \mathcal{E}.fn \mid \mathcal{E}.fn = e \mid p.fn = \mathcal{E}\\
& \mid \mathcal{E}.mn(e^*) \mid p.mn(v^*, \mathcal{E}, e^*) \mid \mathtt{let}\ var = \mathcal{E}\ \mathtt{in}\ e\\
& \mid \mathcal{E}.\mathtt{start}() \mid \mathtt{synchronized}\ \mathcal{E}\ \mathtt{in}\ e \mid \mathtt{insync}\ p\ \mathtt{in}\ \mathcal{E} \mid (t)\mathcal{E}
\end{aligned}$$

State Transition Rules:

[Red New]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[\mathtt{new}\ c()\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma\left[(\!|db|\!)_c^{\mathsf{unlocked}}/p\right], T \cdot (\tau, \mathcal{E}\left[p\right]) \cdot T' \right\rangle$$
$$\text{where } P; c \vdash_{init} db \text{ and } p \notin dom(\sigma)$$

[Red Read]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[p.fn\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[v\right]) \cdot T' \right\rangle$$
$$\text{where } \sigma(p) = (\!|\ldots, fn = v, \ldots|\!)_c^m$$

[Red Assign]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[p.fn = v\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma\left[v/p.fn\right], T \cdot (\tau, \mathcal{E}\left[v\right]) \cdot T' \right\rangle$$

[Red Call]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[p.mn(v_{1\ldots n})\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma, T \cdot \left(\tau, \mathcal{E}\left[e\left[v_i/{x_i}^{i\in\{1\ldots n\}}, p/\mathtt{this}\right]\right]\right) \cdot T' \right\rangle$$
$$\text{where } P; \emptyset \vdash t\ mn(t_i\ {x_i}^{i\in\{1\ldots n\}})\{e\} \in c \text{ and } \sigma(p) = (\!|db|\!)_c^m$$

[Red Let]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[\mathtt{let}\ var = v\ \mathtt{in}\ e\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[e\left[v/var\right]\right]) \cdot T' \right\rangle$$

[Red Sync]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[\mathtt{synchronized}\ p\ \mathtt{in}\ e\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma\left[(\!|db|\!)_c^{\mathsf{locked}}/p\right], T \cdot (\tau, \mathcal{E}\left[\mathtt{insync}\ p\ \mathtt{in}\ e\right]) \cdot T' \right\rangle$$
$$\text{where } \sigma(p) = (\!|db|\!)_c^{\mathsf{unlocked}}$$

[Red InSync]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[\mathtt{insync}\ p\ \mathtt{in}\ v\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma\left[(\!|db|\!)_c^{\mathsf{unlocked}}/p\right], T \cdot (\tau, \mathcal{E}\left[v\right]) \cdot T' \right\rangle$$
$$\text{where } \sigma(p) = (\!|db|\!)_c^{\mathsf{locked}}$$

[Red Start]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[p.\mathtt{start}()\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[\mathtt{null}\right]) \cdot T' \cdot (\tau', p.\mathtt{run}()) \right\rangle$$
$$\text{where } \tau \neq \tau' \text{ and } \tau' \text{ does not appear in } T \cdot T'$$

[Red Cast]

$$P \vdash \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[(t)\ v\right]) \cdot T' \right\rangle \hookrightarrow \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[v\right]) \cdot T' \right\rangle$$
$$\text{where } \sigma(p) = (\!|db|\!)_c^m \text{ with } P; \emptyset \vdash c <: t \text{ and } v = \mathtt{null}$$

Figure 8: The operational semantics for ConcurrentJava$^+$

- Points-to and call-graph analyses are sound, and points-to analysis is independent from concurrency analysis, i.e. the function i_1 in definitions of the functions *inst* and *inst'* is an identity function and does nothing.

- Points-to and call-graph information is available through all steps of evaluation. More precisely, let $S = \langle \sigma, (\mathtt{main}, e_1), \ldots, (\tau_n, e_n)\rangle$ be an arbitrary state from an execution of a program. If $[e_0.mn(e^*)]_{\ell_r}^{\ell_c}$ is a method call site, and is a subexpression of any of e_1 through e_n, then $callees(\ell_c)$ equals the set of all method names that may be called at the states to which S makes a transition (the same assumption is made for thread creation sites). Let e be a subexpression of any of e_1 through e_n, then $mayPointsTo(e)$ equals the set of all abstract objects (heap-shape graph nodes) which correspond to object instantiation sites that the object calculated for e (in a state to which S makes a transition) may be created at any of the sites. In case of addresses, the function *mayPointsTo* is one-to-one, and maps an address (identifier of a concrete object) to a singleton set containing a (unique) synthesized abstract location.

- *Object abstraction function*, α, given an object, maps the object to an abstract location. We suppose that the mapping is one-to-one. The abstract object for an address p, in a state with object store σ, may be computed in a such way that the following condition holds.

$$mayPointsTo(p) = \{\alpha(\sigma(p))\}$$

- Evaluation proceeds without considering labels. Once we want to ascribe an abstract execution to a state (we shall see shortly what it does mean), we (recursively) label expressions in a state. We suppose that applying the function *labMult* on any of such labels results in 1.

- A precalculated method summary exists for each method of the program under execution.

- In order to analyze an expression, we analyze a synthesized method for the expression. We analyze the method by using *IntAnalyze*, and, if needed, instantiate the resulting abstract execution through the function *inst''*. The function *inst''* is defined to be

$$(E', \mapsto') = inst''(\tau^m, (E, \mapsto)) = i_3''(\tau^m, i_2'(\tau, (E, \mapsto))),$$

where the functions i'_2, i''_3 are defined as follows.

$$i'_2(\tau, (E, \mapsto)) = (E\left[\kappa(\tau s, \dots)/\kappa(s, \dots)\right], \mapsto \left[\kappa(\tau s, \dots)/\kappa(s, \dots)\right]),$$

$$\begin{aligned} &i''_3(\tau^m, (E, \mapsto)) = \\ &\quad (E\left[\kappa(\dots, \tau s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right], \mapsto \left[\kappa(\dots, \tau s^{m \otimes m'}, \dots)/\kappa(\dots, s^{m'}, \dots)\right]). \end{aligned}$$

Note that there is no constraint on s. Since the name of the synthesized method is different from the name of any method of the program, and the body expression of the method is unaltered, the method may not be considered to be recursively defined.

- Let $S = \langle \sigma, (\mathtt{main}, e_1), \dots, (\tau_n, e_n) \rangle$ be a state from an execution of the program P with method summaries ms. The abstract execution *ascribed* to the state is defined in the following way.

  $$\bigsqcup \{ \mathit{inst}''(\tau_i, \mathit{IntAnalyze}(P, c.mn, ms)(\bullet)(\mathit{final}(e_i))) \mid 1 \leq i \leq n \wedge \tau_1 = \mathtt{main} \}.$$

  Where $c.mn$ is the name of synthesized method for the epression e_i. We represent the ascription of an abstract execution η to a state S by using the notation $S :: \eta$. Let $S :: \eta$, then we say that S is safe if η is safe.

Definition 1. *Let e be an expression, in the program P, with free variables $v_1, \dots, v_n$ of types $t_1, \dots, t_n$, respectively. Then method c.mn is called the synthesized method for e, if we assume*

$$\texttt{class}\ c\{t\ mn(t_1\ v_1, \dots, t_n\ v_n)\{e\}\}$$

is in in P, where c or mn are fresh identifiers with regard to those that used to occur in P.

Definition 2. *Let $\left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[[p.fn]^\ell\right]) \cdot T' \right\rangle$ be a state. Then the event $R(\tau\ell, \tau^1, \{\alpha(\sigma(p))\})$ is the access event corresponding to the thread τ. Similarly, let $\left\langle \sigma, T \cdot (\tau', \mathcal{E}\left[[p.fn = v]^\ell\right]) \cdot T' \right\rangle$ be state. Then the event $W(\tau'\ell, \tau'^1, \{\alpha(\sigma(p))\})$ is the access event corresponding to the thread τ'.*

It is obvious that an access event corresponding to a thread, as described in the above definition, equals to that abstract event obtained through application of the functions $\mathit{inst}(\tau)$ or $\mathit{inst}'(\tau)$ on the event produced by the transition function for expressions of the form $[p.fn]^\ell$ or $[p.fn = v]^\ell$ for the concurrency analysis given in Section 3.3.

Definition 3. *Restriction of an abstract execution $\eta = (E, \mapsto)$ to a thread identifier τ^m is denoted by $\eta \mid_{\tau^m}$, and is defined to be the sequence $< e_1, \dots, e_k >$, where $e = \kappa(id, \tau^m, L) \in E$ if and only if there exists an $i \in \mathbb{N}$, such that $e_i = e$. Furhtermore, for each $1 \leq i < k$ we have $e_i \mapsto e_{i+1}$. We call the restriction of an abstract execution to a thread identifier, a trail of abstract events.*

Definition 4. *Let T_1, T_2 be two trails of abstract events. We say that the trails are equivalent, and represent the fact by using the notation $T_1 \equiv T_2$, if they are of the same length, and for each abstract event $e = \kappa(id, \tau^m, L)$ in T_1, and $e' = \kappa'(id', \tau'^{m'}, L')$ in T_2 we have $\kappa = \kappa'$, $m = m'$, and $L = L'$.*

It is clear that the relation $\equiv$ is an equivalence relation. Two equivalent trails of abstract events denote the same sequence of actions, hence parital and augmented effect of the actions are the same. In other words, two equivalent trails of abstract events are two sequences of events that are the same up to renaming of identifiers.

We may extend the notion of equivalence of trails of abstract events to the notion of isomorphism of abstract executions. The second component in an abstract execution, i.e. the relation $\mapsto$, imposes a structure between events in the abstract execution. Hence, the relation imposes a structure between the trails of events in that abstract execution. Thus, informally, two abstract executions are isomorphic if they consist of equivalent trails of abstract events, and the imposed structures between trails of abstract events in the abstract executions are the same. It is obvious that two isomorphic abstract executions denote the same set of executions.

Before presenting a precise definition of the notion of isomorphism of abstract executions, we need to give a number of auxiliary definitions. Let $T =< e_1, \dots, e_k >$ be a trail of abstract events, $\mathit{eventsOf}(T)$ is defined to be the set of all events occuring in T, i.e. the set $\{e_1, \dots, e_k\}$. Given an abstract execution $\eta = (E, \mapsto)$, $\mathit{threadsOf}(\eta) = \{\tau \mid \kappa(id, \tau^m, L) \in E\}$. We say that abstract executions $\eta_1 = (E_1, \mapsto_1)$, $\eta_2 = (E_2, \mapsto_2)$ are isomorphic with regard to thread identifiers $\tau_1^{m_1}$, $\tau_2^{m_2}$, respectively, if assuming $T_1 = \eta_1 \mid_{\tau_1^{m_1}}$, $T_2 = \eta_2 \mid_{\tau_2^{m_2}}$, the following conditions hold:

1. $T_1 \equiv T_2$, and

2. For each $e_1 \in \mathit{eventsOf}(T_1)$ with $e_1 = \kappa_1(id_1, \tau_1^{m_1}, L_1)$, where $e_1 \mapsto_1 e_1'$ with $e_1' = \kappa_1'(id_1', \tau_2^{m_2}, L_1')$ and $\tau_1^{m_1} \neq \tau_2^{m_2}$, we have $e_2 = f(e_1, T_1, T_2) = \kappa_2(id_2, \tau_3^{m_3}, L_2)$ and $e_2 \mapsto_2 e_2'$ with $e_2' = \kappa_2'(id_2', \tau_4^{m_4}, L_2')$ and $\tau_3^{m_3} \neq \tau_4^{m_4}$, and have $(\eta_1, \tau_2^{m_2}) \approx (\eta_2, \tau_4^{m_4})$.

The function f, in this definition, is defined in the following manner. Let T_1, T_2 be two trails of abstract events, such that $T_1 \equiv T_2$ and $e \in \mathit{eventsOf}(T_1)$. For each $n \in \mathbb{N}$ s.t. $T_1(n) = e$ implies $f(e, T_1, T_2) = T_2(n)$. We represent the isomorphism of η_1, η_2 with regard to the thread identifiers $\tau_1^{m_1}$, $\tau_2^{m_2}$ by using the notation $(\eta_1, \tau_1^{m_1}) \approx (\eta_2, \tau_2^{m_2})$.

Definition 5. *We say that two abstract executions η_1, η_2 are isomorphic, and write $\eta_1 \simeq \eta_2$, if the following conditions hold:*

1. $\forall \tau_1^{m_1} \in \mathit{threadsOf}(\eta_1).\ \exists \tau_2^{m_2} \in \mathit{threadsOf}(\eta_2).\ (\eta_1, \tau_1^{m_1}) \approx (\eta_2, \tau_2^{m_2}) \wedge (\eta_2, \tau_2^{m_2}) \approx (\eta_1, \tau_1^{m_1})$*, and*
2. $\forall \tau_2^{m_2} \in \mathit{threadsOf}(\eta_2).\ \exists \tau_1^{m_1} \in \mathit{threadsOf}(\eta_1).\ (\eta_1, \tau_1^{m_1}) \approx (\eta_2, \tau_2^{m_2}) \wedge (\eta_2, \tau_2^{m_2}) \approx (\eta_1, \tau_1^{m_1})$.

It is apparent that the relation defined in Definition 5 is an equivalence relation. Note that the structure of two isomorphic abstract events (view as two graphs) is the same, and for each trail of abstract events from one abstract execution, there is an equivalent trail of abstract events from the other abstract execution, and vice versa. Hence, a little thought reveals that one the abstract executions is safe if and only if the other is safe.

By reviewing the definitions for the instantiation functions *inst*, *inst′*, and *inst″*, we see that the functions accomplish instantiation through prefixing event and thread identifiers of each abstract event in an abstract execution with a symbol. It is clear that the functions, during instantiation, *do not* alter the structure of abstract executions. Thus, we may have the following lemma.

Lemma 1. *Let η be an abstract execution. Given an arbitrary label ℓ, we have:*

$$\eta \simeq \mathit{inst}(\ell, \eta),$$
$$\eta \simeq \mathit{inst}'(\ell, \eta).$$

Furthermore, given an arbitrary thread identifier τ^m, we have $\eta \simeq \mathit{inst}''(\tau^m, \eta)$.

Definition 6. *Let $S = \langle \sigma, T \cdot (\tau, e) \cdot T' \rangle$ be a state, in which e is an expression of the form $\mathcal{E}\left[[p.fn]^\ell\right]$ or $\mathcal{E}\left[[p.fn = v]^\ell\right]$. The set of all locks acquired by τ while accessing p is defined as follows.*

$$L_\tau^p = \{p' \mid \ldots \texttt{insync}\ p'\ \texttt{in} \ldots [e']^\ell \cdots \in e\},$$

where e' is the expression filling the hole in e.

Lemma 2. *Let S be an arbitrary state in which two distinct threads τ, τ' access p, and $S :: \eta$. Then the two access events do not have ord_η relationship.*

Proof. We suppose otherwise, that is to say, assume the two events have ord_η relationship with each other. Then, according to the definition of the relation, there are two cases to consider:

1. The events belong to a single thread, i.e. have the same thread identifiers,
2. The events are from different threads, and are ordered through monitor operations.

The first case immediately contradicts the hypothesis of the theorem stating that the events belong to different threads. For the second case we need to examine two subcases. First, the events are themselves (matching) monitor operations. Since we have assumed that the events are access events, this case would be a contradiction. Second, the events are two access events, and are surrounded by matching monitor events from each thread. Which means that the threads τ, τ', while accessing p, acquire at least a common lock. More precisely, for the state S, we have $L_\tau^p \cap L_{\tau'}^p \neq \emptyset$. By using contrapositive of Lemma 3, we conclude that the two threads does not commit any accesses, which contradicts the assumption that the threads access p. Therefore, the two access events from the threads τ, τ' does not have ord_η relation with each other. □

Lemma 3. *Let S be an arbitrary state, and τ, τ' be two distinct threads in the state. If the threads access the addresses p_1, p_2, then $L_\tau^{p_1} \cap L_{\tau'}^{p_2} = \emptyset$.*

Proof. We suppose that the intersection of the sets is non-empty, i.e. there is a $p \in L_\tau^{p_1} \cap L_{\tau'}^{p_2}$. Thus, there are states S', S'', such that

$$S' = \langle \sigma_1, T_1 \cdot (\tau, \mathcal{E}\left[\texttt{synchronized}\ p\ \texttt{in}\ e\right]) \cdot T_2 \rangle \hookrightarrow^* S_i \hookrightarrow^* S,$$

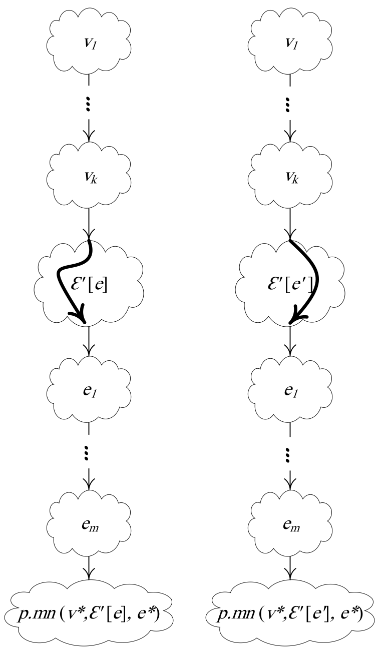


Figure 9: A pictorial depiction of $\eta_{p.mn(v^*,\mathcal{E}'[e_1],e^*)}$ (left) and $\eta_{p.mn(v^*,\mathcal{E}'[e_2],e^*)}$ (right)

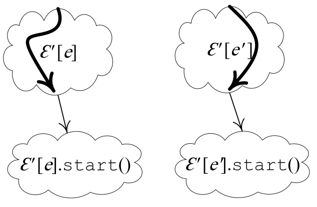


Figure 10: A pictorial depiction of $\eta_{\mathcal{E}'[e_1].\texttt{start()}}$ (left) and $\eta_{\mathcal{E}'[e_2].\texttt{start()}}$ (right)

and

$$S'' = \langle \sigma_2, T_1' \cdot (\tau', \mathcal{E}\,[\texttt{synchronized}\ p\ \texttt{in}\ e']) \cdot T_2' \rangle \hookrightarrow^* S_j \hookrightarrow^* S.$$

Furthermore, neither of the states S_i nor S_j are of the form $\langle \sigma_3, T_3 \cdot (\tau, \mathcal{E}\,[\texttt{insync}\ p\ \texttt{in}\ v]) \cdot T_4 \rangle$ and $\langle \sigma_4, T_3' \cdot (\tau', \mathcal{E}\,[\texttt{insync}\ p\ \texttt{in}\ v]) \cdot T_4' \rangle$ respectively. We examine the case of S'; the other case is handled similarly.

Without loss of generality, we assume $S' \hookrightarrow^* S''$. According to the reduction rules [Red Sync], [Red InSync], given in Figure 8, if in the state S' it is the case that $\sigma(p) = (\!|db|\!)_c^{\mathsf{unlocked}}$, the state would make transition to a state within which lock-state of p is locked. Thereby preventing evaluation of any expression of the form `synchronized` p `in` e' during the rest of the execution—unless the lock is released by the thread, of course. Hence, we have

$$p \notin L_{\tau'}^{p_2}. \tag{1}$$

On the other hand, if $\sigma(p) = (\!|db|\!)_c^{\mathsf{locked}}$, then, according to the precondition of the reduction rule [Red Sync], the thread τ would be prohibited from further evaluation, which leads to

$$p \notin L_{\tau}^{p_1}. \tag{2}$$

Either of (1) or (2) leads to a contradiction. Thus, we have the desired result. □

In what follows, in order to simplify presentation, for an expression e in program P with methods summary ms, we may abbreviate $IntAnalyze(P, ms, c.mn)(\bullet)(final(e))$ where $c.mn$ is the name of the synthesized method for e, as η_e.

Lemma 4. *Let $\mathcal{E}$ be an evaluation context and e_1, e_2 be two expressions. Then $\eta_{e_1} \simeq \eta_{e_2}$ and $(\eta_{e_1}, \Lambda^?) \approx (\eta_{e_2}, \Lambda^?)$ implies $\eta_{\mathcal{E}[e_1]} \simeq \eta_{\mathcal{E}[e_2]}$ and $(\eta_{\mathcal{E}[e_1]}, \Lambda^?) \approx (\eta_{\mathcal{E}[e_2]}, \Lambda^?)$.*

Proof. We proceed by induction on the structure of evaluation contexts, which is defined in Figure 8. We will just inspect two of non-trivial forms.

$\mathcal{E} = p.mn(v_1, \ldots, v_k, \mathcal{E}', e_1, \ldots, e_m)$: By replacing the hole with e_1, e_2 we get the expressions $\mathcal{E}\left[e_1\right]$, $\mathcal{E}\left[e_2\right]$, respectively. Figure 9 does pictorially show a sketch of the abstract executions $\eta_{\mathcal{E}[e_1]}$, $\eta_{\mathcal{E}[e_2]}$ as two graphs. The structure of the graphs may be justified according to the way the control-flow graph for the expressions $\mathcal{E}\left[e_1\right]$, $\mathcal{E}\left[e_2\right]$ is constructed. In Figure 9 (and also in Figure 10), clouds represent the parts of the abstract execution corresponding to the sub-expression written on them. The direction of the arrows represent the direction of the relation $\mapsto$, i.e. the second compartment of the abstract execution. A thick line in the graph represents a path of events with thread identifier $\Lambda^?$.

Due to induction hypothesis, we have:

$$\eta_{\mathcal{E}'[e_1]} \simeq \eta_{\mathcal{E}'[e_2]} \ \wedge \ (\eta_{\mathcal{E}'[e_1]}, \Lambda^?) \approx (\eta_{\mathcal{E}'[e_2]}, \Lambda^?) \tag{3}$$

As we can see in Figure 9, the two abstract executions, except for the parts corresponding to $\eta_{\mathcal{E}'[e_1]}$ and $\eta_{\mathcal{E}'[e_2]}$, share the same structure. By referring to the transition functions of non-modular analysis we realize that the abstract executions, excluding the aforementioned parts, are the same up to renaming of identifiers. So, considering (3), we may conclude the desired result.

$\mathcal{E} = \mathcal{E}'.\texttt{start}()$: Similar to the previous case, Figure 10 shows a sketch of the abstract executions $\eta_{\mathcal{E}[e_1]}$, $\eta_{\mathcal{E}[e_2]}$. According to the induction hypothesis, we have:

$$\eta_{\mathcal{E}'[e_1]} \simeq \eta_{\mathcal{E}'[e_2]} \ \wedge \ (\eta_{\mathcal{E}'[e_1]}, \Lambda^?) \approx (\eta_{\mathcal{E}'[e_2]}, \Lambda^?) \tag{4}$$

The identicalness of the lower parts of the graphs shown in Figure 10, together with (4), yields our desired result. □

Theorem 1 (Preservation)**.** *Let S be a safe state, and suppose $S \hookrightarrow S'$. Then S' is safe.*

Proof. We proof the theorem by an induction on the structure of $S \hookrightarrow S'$, which is shown in Figure 8. For all of the cases, the result follows directly from induction hypothesis. The only rules that deserve debate are that of method call and of thread creation.

The rule [**Red Call**] Assume that the state $S = \langle \sigma, T \cdot (\tau, e_{call}) \cdot T' \rangle$, where the expression e_{call} is supposed to be $\mathcal{E}\left[\left[p.mn(v_1, \ldots, v_m)\right]_{\ell_r}^{\ell_c}\right]$, is safe. Let $\sigma(p) = (\!|db|\!)_c^m$, and $declOf(c.mn)$= $t\ mn(t_1\ x_1, \ldots, t_m\ x_m)\{e\}$. We are going to prove that the state $S' = \langle \sigma, T \cdot (\tau, e_b) \cdot T' \rangle$, where $e_b = \mathcal{E}\left[e\left[v_k/x_k{}^{k \in \{1,\ldots,m\}}, p/\texttt{this}\right]\right]$, is also safe.

We observe that the only difference between S, S' is in the sequence of threads, in particular, is in the expression that fills the hole in the expression corresponding to the thread τ. In order to obtain the abstract execution $IntAnalyze(P, ms, c'.mn')(\bullet)(\ell_r))$, where $c'.mn'$ is the name of synthesized method for the expression $[p.mn(v_1, \ldots, v_m)]_{\ell_r}^{\ell_c}$, since $c'.mn'$ is chosen to be fresh in P (hence, it is not recursively defined with any of the methods already existing in P), and since we have assumed that there is a method summary in ms for each method of P, the transition function for method call sites uses precalculated method summary for $c.mn$. Thus, according to Lemma 1, $IntAnalyze(P, ms, c'.mn')(\bullet)(\ell_r)$, and $ms(c.mn)$ are isomorphic. A little thought reveals that the two abstract executions are even identical up to renaming of the identifiers. Thus, we also have the following fact.

$$(IntAnalyze(P, ms, c'.mn')(\bullet)(\ell_r), \Lambda^?) \approx (ms(c.mn), \Lambda^?) \tag{5}$$

Furthermore, it is clear that $IntAnalyze(P, ms, c''.mn'')(\bullet)(final(e))$, where e is the body of $c.mn$ and $c''.mn''$ is the name of synthesized method for e, is isomorphic to $ms(c.mn)$. Similar to above, it is easy to check that the abstract executions are identical up to renaming of the identifiers. Therefore, we have the following fact, as well.

$$(IntAnalyze(P, ms, c''.mn'')(\bullet)(final(e)), \Lambda^?) \approx (ms(c.mn), \Lambda^?) \tag{6}$$

Thus, the transitive and reflexive properties of the relation $\simeq$ implies that the abstract executions $IntAnalyze(P, ms, c'.mn')(\bullet)(\ell_r))$ and $IntAnalyze(P, ms, c''.mn'')\ (\bullet)(final(e))$ are isomorphic. So, according to Lemma 4, (5) and (6) leads to $\eta_{e_{call}} \simeq \eta_{e_b}$. This, in turn, yields our desired result, i.e. S' is safe.

The rule [**Red Start**] Suppose that the state $S = \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[\left[p.\texttt{start}()\right]^{\ell}\right]) \cdot T' \right\rangle$ is safe. Let $S \hookrightarrow S'$, where $S' = \left\langle \sigma, T \cdot (\tau, \mathcal{E}\left[\texttt{null}\right]) \cdot T' \cdot (\tau', [p.\texttt{run}()]_{\ell_r}^{\ell_c}) \right\rangle$. We are going to show that S' is also safe.

Since the the transition functions of the analysis do not produce any event in the case of the value `null`, according to the induction hypothesis, we have $S'' = \langle \sigma, T \cdot (\tau, \mathcal{E}\left[\texttt{null}\right]) \cdot T' \rangle$ safe. Furthermore, induction hypothesis, implies that

$$inst''(\tau^1, IntAnalyze(P, ms, c'.mn')(\bullet)(\ell)), \tag{7}$$

where $c'.mn'$ is the name of synthesized method for $[p.\texttt{start}()]^{\ell}$, is safe. Since the events, and the ordering among them, produced for the expression $p.\texttt{run}()$ in the thread τ' are the same as the ones produced in (7) up to renaming of identifiers, we observe that $inst''(\tau'^1, IntAnalyze\ (P, ms, c''.mn'')(\bullet)(\ell_r))$, where $c''.mn''$ is the name of synthesized method for the expression $[p.\mathtt{run}()]^{\ell_c}_{\ell_r}$, is also safe. Thus, considering the induction hypothesis, the sate S' is safe. □

Theorem 2 (Safety)**.** *Let S be an arbitrary sate with $S :: \eta$. If η is safe, then η does not contain conflicting accesses, i.e. the state does not have a data race error.*

Proof. Suppose otherwise, and assume the state, given $\tau \neq \tau'$, is in one of the following forms.

- $S = \langle \sigma, T \cdot (\tau, \mathcal{E}\,[p.fn]) \cdot T' \cdot (\tau', \mathcal{E}\,[p.fn = v]) \cdot T'' \rangle$, or
- $S = \langle \sigma, T \cdot (\tau, \mathcal{E}\,[p.fn = v_1]) \cdot T' \cdot (\tau', \mathcal{E}\,[p.fn = v_2]) \cdot T'' \rangle$.

The subexpressions $p.fn$ and $p.fn = v$, in a control flow graph for the expressions $\mathcal{E}\,[p.fn]$ and $\mathcal{E}\,[p.fn = v]$, are represented by using two nodes and the concurrency analysis produces two abstract events for the nodes. According to the transition functions for dereference and assignment statements, given in Section 3.3, instantiating the events produced by the functions using $inst''(\tau^1)$, $inst''(\tau'^1)$ results in events that equal with the access events corresponding to the threads up to renaming of the identifiers. Therefore, the two events are conflicting. Furthermore, by using Lemma 2 we conclude that the events do not have ord_η relationship with each other. So, η is not safe, which is a contradiction. □

Definition 7. *An execution of the program $P = defn^*e$ is defined to be: (1) a finite sequence S_1, S_2, ..., S_k of states, where $S_1 = \langle \emptyset, (\mathtt{main}, e) \rangle$, $S_k = \langle \sigma, (\mathtt{main}, v_1) \cdots (\tau_n, v_n) \rangle$, and $S_i \hookrightarrow S_{i+1}$ for $0 < i < k$; or (2) an infinite sequence S_1, S_2, ... of states, where $S_1 = \langle \emptyset, (\mathtt{main}, e) \rangle$ and for each $i > 0$ we have $S_i \hookrightarrow S_{i+1}$.*

Corollary 1. *Let $P = defn^*e$ be a program, and η be the abstract execution obtained for the final label of e, i.e. $\eta = \eta_e$. Suppose that η is safe, and $\langle S_i \rangle$ be an arbitrary execution of the program. Then, each state S_i in the execution is free of conflicting accesses.*

Proof. We divide the proof of this theorem into two parts. First, we prove the result for finite executions, next we show that the results still hold for infinite executions. According to the Definition 7, we observe that the first element of every execution is the state $S_1 = \langle \emptyset, (\mathtt{main}, e) \rangle$. Hypothesis of the theorem, stating that η (the abstract execution obtained for the entry point of the program P) is safe, implies that the state S_1 is also safe. Now, we use a mathematical induction on the size of finite executions to complete first part of the theorem.

The execution is of the form $\langle S_1 \rangle$: It is clear that S_1 is safe.

Induction hypothesis: For an arbitrarily chosen $k > 1$, all of the states in the execution $\langle S_1, \ldots, S_k \rangle$ are safe.

Inductive step: Let $\langle S_1, \ldots, S_k, S_{k+1} \rangle$ be an execution of the program. According to inductive hypothesis we see that all of the states in $\langle S_1, \ldots, S_k \rangle$, including S_k, are safe. Since $S_k \hookrightarrow S_{k+1}$, by using the Theorem 1, we conclude that S_{k+1} is also safe.

We use proof by contradiction to handle second part of the theorem. Let $\langle S_1, S_2, \ldots \rangle$ be an (infinite) execution of the program, where S_k is unsafe for at least a $k > 0$. Without loss of generality, suppose that k is the index of the first unsafe state. Note that k should be greater than 1, because we have already verified that S_1 is safe. Since S_1 is safe, and $S_1 \hookrightarrow^* S_k$, through k-times application of the Theorem 1, we reach a contradiction. So, all of the states in $\langle S_1, S_2, \ldots \rangle$ are safe.

Thus far, we have shown that, given $\langle S_i \rangle$ be a possibly infinite execution, each S_i in the execution is safe. Through Theorem 2, we conclude that each S_i is free of conflicting accesses, and hence free of data races. □

F An Analysis Order for Methods

As we have seen in Section 3.4, the transition function for method call sites in the modular concurrency analysis presupposes the condition (7) to hold. We claim that through an appropriate order in the analysis of the methods in a program, we may satisfy the precondition. In this section, we present an algorithm that builds such an ordering. We have proved correctness of the algorithm.

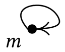
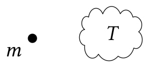
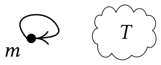
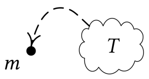
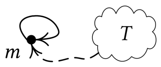
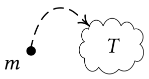
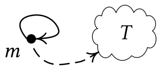
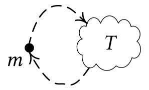
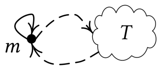
For ease of presentation, we need to give a couple of definitions. Let $m \in Methods_P$ be a method of a program P. *Recursive group* of m, denoted by $recGroup(m)$, is defined to be the set of methods in $Methods_P$ that are defined mutually recursive in terms of m. More precisely, $recGroup(m) = \{m' \mid m' \in Methods_P \wedge (m, m') \in SCC\}$. We define recursive groups for a set $M \subseteq Methods_P$ of methods, in terms of $recGourp$, to be $RecGroups(M) = \bigcup\{recGroup(m) \mid m \in M\}$.

The algorithm is presented in Figure 11. Note that, for ease of presentation, we have used the procedure DoAnalyze(m) to abbreviate: (1) analysis of m by using the modular analysis given in Section 3.4, (2) update of ms, so that the function maps m to the result calculated at step (1).

```
Input:   A program $P$, the set $Methods_P$, and $ms$
Output:  All methods in $Methods_P$ are analyzed.
 1: $M = \emptyset$
 2: $N = Methods_P$
 3: do
 4:     for each $m \in N$ do
 5:         if $mayCall(m) \subseteq M$ then
 6:             DoAnalyze($m$)
 7:             $M = M \cup \{m\}$
 8:     for each $R \in RecGroups(N)$ do
 9:         for each $m \in R$ do
10:             if $mayCall^*(m) \subseteq (R \cup M)$ then
11:                 DoAnalyze($m$)
12:                 $M = M \cup \{m\}$
13:     $N = N - M$
14: while $N \neq \emptyset$
```

Figure 11: An algorithm computing an order for analysis of methods in a program, while satisfying precondition (7) of Section 3.4

Table 1: Different states for shape of a call-graph, when we add the method m to it. The part represented using a cloud, is a subgraph of the call-graph which deos not contain m. A solid edge denoted a self-recurrence of m, and a dashed edge denotes a non-empty collection of caller-callee relationship(s) between m and the method(s) in the subgraph represented by T. Direction of an edge describes which method(s) calls the other(s).

(1)		(2)	
(3)		(4)	
(5)		(6)	
(7)		(8)	
(9)		(10)	

Theorem 3. *The following conditions hold for the algorithm of Figure 11:*

- *Condition (7) from Section 3.4 is satisfied before each call to the procedure* DoAnalyze*,*
- *The algorithm always terminates.*

Proof. The **if** conditions at lines 5 and 10, quarantee that whenever the algorithm sets forth to use the procedure DoAnalyze in order to analyze the method m, either all of the methods that m calls are analyzed beforehand, hence there are method summaries for the methods, or the methods are in a recursive group with m, so there is no need for a method summary.

For the second part of the theorem, we proceed by course of value induction on the size of the set $Methods_P$ to show that the **do-while** loop of lines 3-14 does indeed terminate for all programs P with the set of methods $Methods_P$.

$|Methods_P| = 1$ In this case the call graph of the program is of the form illustrated in parts (1), (2) of Table 1. In case (1), since $mayCall(m) = \emptyset$, first loop of the algorithm (lines 4-7) handles the graphs. In case (2), since $R = \{m\}$ is the only recursive group for the program, we have $mayCall^*(m) \subseteq R$. Hence second loop of the algorithm (lines 8-12) cops with the problem. In either case, once the method m is added to the set M the algorithm halts.

Indunction hypothesis The algorithm terminates for each set $Methods_P$ with a cardinality less than k, where $k > 1$.

$|Methods_P| = k$ In order to complete the proof, we need to examine each possible shape for the call-graph depicted in parts (3)-(10) of Figure 1, in which a cloud, named T is a subgraph of the whole call-graph whose size is less than k. In either of cases (3) or (4), according to the induction hypothesis, the methods in the subgraph T are analyzed. In case (3), the method m is handled by the first loop of the algorithm, and in case (4), it is handled by the second loop of the algorithm. In either cases, once the method m is added to M, the algorithm halts.

In cases (5) and (6), the analysis of m is handled by first and second loops of the algorithm, respectively, and the values of M and N changes to $\{m\}$ and $Methods_P - \{m\}$, accordingly. Now, the algorithm is faced with a set of methods with a size less than k, so induction hypothesis implies that the algorithm terminates.

In cases (7) and (8), the algorthim first finishes the subgraph T of the call-graph and handles m using its first and second loops, respectively. Thus, the algorithm also terminates in these cases.

From cases (9) and (10), we discuss (9); the other can be inspected in a similar way. First, suppose the method m has a SCC relationship with all of the methods that it calls or is called by. In such a case m and the methods fall in to a single recursive group, that is to say, the set of methods of the subgraph T is partitioned into two set T_{SCC}, T'_{SCC}, where the former is the set of methods (including m) that have an SCC relationship with m, and the latter is those methods that has not such a relationship with m. Obviously, $|T'_{SCC}| < k$, so the algorithm halts on this subset. The set T_{SCC}, with $|T_{SCC}| \leq k$, is handled through the second loop of the algorithm. When the method m does not have SCC relationship with any of the methods in T, the proof of the algorithm reduces to the proof of cases (5) and (7) which we have formerly discussed. The only uninspected situation is the case in which $T_{SCC} \neq \emptyset$, and some methods of T call m while they have not SCC relationship with it. We represent the set by T_m. Clearly, we have $T_m \neq \emptyset$. Furthermore, we now that the method m calles a number of methods of T that does not have SCC relationship with the methods. We represent the set by m_T, and we observe that $m_T \neq \emptyset$. Obviously, $|m_T| < k$, and since methods in this set are not in a single recursive group with m, the analysis of the methods does not depend on the analysis of m or any of the methods that are in a recursive group with m. So, according to the induction hypothesis, the algorithm halts in this case. After that we would have $M = m_T$, $N = Methods - m_T$. Since m_T is non-empty, the algorithm encounters with a set of methods with a cardinality less than k, and induction hypothesis implies that the algorithm also halts on this input. □

Bibliography

[1] J. Whaley and M. Rinard, “Compositional pointer and escape analysis for java programs,” in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’99, (New York, NY, USA), pp. 187–206, ACM, 1999.

[2] M. Sagiv, T. Reps, and R. Wilhelm, “Solving shape-analysis problems in languages with destructive updates,” *ACM Transactions on Programming Languages and Systems*, vol. 20, pp. 1–50, January 1998.

[3] M. Flatt, S. Krishnamurthi, and M. Felleisen, “Classes and mixins,” in *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, POPL ’98, (New York, NY, USA), pp. 171–183, ACM, 1998.

[4] M. Abadi, C. Flanagan, and S. N. Freund, “Types for safe locking: Static race detection for java,” *ACM Transactions on Programming Languages and Systems*, vol. 28, pp. 207–255, Mar. 2006.