Deadlock detection of Java Bytecode

Abel Garcia and Cosimo Laneve

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus
`cosimo.laneve@unibo.it`

Abstract

This paper presents a powerful technique for deadlock detection of `Java` programs. The technique uses typing rules for extracting infinite-state abstract models of the dependencies among the components of the `Java` intermediate language – the `Java` bytecode. Models are subsequently analysed by means of an extension of a solver that we have defined for detecting deadlocks in process calculi. Our technique is complemented by a prototype verifier that also covers most of the `Java` features.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Deadlocks are common flaws of concurrent programs that occur when a set of threads are blocked because each one is attempting to acquire a lock held by another one. Such errors are difficult to detect or anticipate, since they may not happen during every execution, and may have catastrophic effects for the overall functionality of the software system. At the time of writing this paper, the *Oracle Bug Database*[1] reports more than 40 unresolved bugs due to deadlocks, while the *Apache Issue Tracker*[2] reports around 400 unresolved deadlock bugs. These two databases refer to programs written in `Java`, a mainstream programming language in a lot of domains, such as web and cloud applications, user applications and mobile applications.

The objective of our research is to design and implement a technique capable of detecting potential deadlock bugs of `Java` programs *at compilation time*. This objective is difficult because `Java` has a complex concurrent model: it uses threads that may perform read/write operations over shared variables and whose execution depends on the scheduling strategy implemented in the Java Virtual Machine (`JVM`). In addition, `Java`, being a full-fledged programming language, includes an extensive standard library with lots of features implemented in native language.

To reduce the complexity of our work, we decided to address the `Java` bytecode, namely 198 instructions that are the compilation target of every `Java` application and have a reference semantics that is defined by the `JVM` behaviour. Therefore, it is possible to deliver correctness results without narrowing our original goal or oversimplifying our job. In fact, our technique is similar to those used for demonstrating the correctness of the Bytecode Verifier [19, 6, 15, 12].

In order to present our deadlock detection technique, we isolate a subset of `Java` bytecode, called $\texttt{JVML}_d$, which includes basic instructions for concurrency, such as thread creations, synchronizations, and creations of new objects. The language is defined in Section 3. The technique consists of two stages.

The first stage defines a type system that reconstructs the concurrent behaviour of methods. The key principles are the following ones. Each method has an associated type

[1] http://bugs.java.com/
[2] https://issues.apache.org/jira

that depends on the type of the arguments (the object "`this`" is one argument) and that expresses the *concurrent behaviour*. This "concurrent behaviour" reports (*i*) the *sequence of locks* that has been acquired/released by the method, (*ii*) the *threads created*, and (*iii*) the *methods that have been invoked*. It includes the analysis of aliases that traces the creation of new objects and their copies (because $\texttt{JVML}_d$ instructions may create and copy objects). The alias analysis is performed in a *symbolic way* by using a *finite set of names*: this is a critical part of our technique because methods may create threads and, when methods are either recursive or iterative, the set of created threads may be infinite. In particular, we had to devise finite representatives of (infinite sets of) thread names that are sound with respect to the (deadlock) analysis. Section 2 reports a code that can be written in (a simple extension of) $\texttt{JVML}_d$ and that is problematic as regards deadlock detection. Section 4 describes the type system.

The second stage of our technique defines the analysis of the behavioural model. In fact, the three reports above – (*i*), (*ii*), and (*iii*) – are terms in a modelling language that extend so-called *lams* [11, 10, 14]. Lams are conjunctions and disjunctions of object dependencies and method invocations and the extension has been necessary for modelling `Java` *reentrant locks*. In particular, our dependencies also carry thread names – $(a, b)_t$ means that the thread t, which owns the lock of a, is going to lock b. In `Java`, the lam $(a, a)_t$ is not a circular dependency because it means that t is acquiring the lock of a *twice*. Because of this extension, the algorithm for detecting circularities in lams is different than the one in [10, 14]. We address this issue in Section 5.

Our deadlock detection technique has been prototyped and the verifier is called `JaDA`. While the type system in this paper simply checks static informations, `JaDA` infers the behavioural types from the bytecode (an overview of the algorithm is reported in [9]). Inference is important in practice because it lightens the analysis but, for space limitations, we decided to discuss it in the full paper. However, checking is crucial for type safety, which we address in a (very) concise way in Section 6 (for reviewer's sake, the details appear in the Appendix). It is also worth to notice that `JaDA` includes several features of `JVML`; this has made possible to deliver initial assessments of the tool. Section 7 discusses `JaDA`, its assessments and related work; we conclude in Section 8.

2 Overview of `JVML` and of our technique

Figure 1 reports a `Java` class called `Network` and some of its $\texttt{JVML}_d$ representation. The corresponding `main` method creates a network of `n` threads by invoking `buildNetwork` – say $t_1, \cdots, t_n$ – that are all potentially running in parallel with the caller – say t_0. Every two adjacent threads share an object, which are also created by `buildNetwork`. Every thread t_i locks the two adjacent objects, that are passed as (implicit) arguments of the thread, and terminates – this is performed by the method `takeLocks`. It is well-known that, if the network is circular – the thread t_n is sharing one of its objects with t_0 and if all the threads have a symmetric strategy of locking objects then a deadlock may occur. On the contrary, if the network is not circular, no deadlock will ever occur. Therefore `buildNetwork(n,x,x)` is deadlocked (when $\texttt{n} > 0$) and `buildNetwork(n,x,y)` is deadlock free (assuming $\texttt{x} \neq \texttt{y}$).

The problematic issue of `Network` is that the number of threads is not known statically because `n` is an argument of `main`. This is displayed in the bytecode of `buildNetwork` in Figure 1 by the instruction at address 30 where a new thread is created and by the instruction at address 37 where the thread is started. The recursive invocation that causes the (static) unboundedness is found at instruction 47. Our technique is powerful enough to cope with such

```
class Network{

 public void main(int n){
   Object x = new Object();
   Object y = new Object();
   // buildNetwork(n, x, x); // deadlock
   buildNetwork(n, x, y); //no deadlock
 }

 public void buildNetwork(int n,
               Object x, Object y){
   if (n==0) {
     takeLocks(x,y) ;
   } else {
     final Object z = new Object() ;
     Thread t = new Thread(){
       public void run(){
         takeLocks(x,z) ;
     }} ;
     t.start();
     this.buildNetwork(n-1,z,y) ;
   }
 }

 public void takeLocks(Object x,
                      Object y){
   synchronized(x){ synchronized(y){ } }
 }
}
```

```
public void buildNetwork(int n, Object x, Object y)
   0  iload_1          //n
   1  ifne 13
   4  aload_0          //this
   5  aload_2          //x
   6  aload_3          //y
   7  invokevirtual 24 //takeLocks(x, y):void
  10  goto 50
  13  new 3
  16  dup
  17  invokespecial 8  //Object()
  20  astore 4         //z
  22  new 26
  25  dup
  26  aload_0          //this
  27  aload_2          //x
  28  aload 4          //z
  30  invokespecial 28 //Network$1(this, x, z)
  33  astore 5         //thr
  35  aload 5          //thr
  37  invokevirtual 31 //start():void
  40  aload_0          //this
  41  iload_1          //n
  42  iconst_1
  43  isub
  44  aload 4          //z
  46  aload_3          //y
  47  invokevirtual 36 //buildNetwork(n-1, z, y):void
  50  return
```

Figure 1 Java Network program and corresponding bytecode (only the `buildNetwork` method). Comments in the bytecode give information of the objects used and/or methods invoked in each instruction

problems and predict the faulty behaviour in case of the invocation `buildNetwork(n,x,x)` and the correct behaviour if the invocation is `buildNetwork(n,x,y)`. The technique works as follows. It infers abstract methods' behaviors by computing types, called *lams*, of their bytecode bodies. These lams abstract each bytecode instruction by dropping the *non-relevant* information for the deadlock analysis (e.g. operations on integer variables). In practice, the relevant operations for deadlock analysis are: locking operations (`monitorenter` and `monitorexit` instructions), thread spawning operations, function invocations and objects' structures. Thereafter the abstract model is analysed by a solver.

3 The language $\texttt{JVML}_d$

$\texttt{JVML}_d$ is a restriction of `JVML` that includes basic constructs and instructions for concurrency [3]. In $\texttt{JVML}_d$, a program is a collection of *class files* whose methods have bodies written in $\texttt{JVML}_d$ bytecode. This bytecode is a partial map from *addresses* $\textsc{Addr}$ to instructions. Addresses, ranged over L, L', $\cdots$, are intended to be nonnegative integers and we use the function $L+1$ that returns the least address that is strictly greater than L. When P is a bytecode, we write $dom(P)$ to refer to its domain (the set of addresses) and we assume that $0 \in dom(P)$ for every bytecode P.

We use a number of *names*: for classes, ranged over by $\texttt{C}$, $\texttt{D}$, $\cdots$, for fields, ranged over by $\texttt{f}$, $\texttt{f}'$, $\cdots$, for methods, ranged over by $\texttt{m}$, $\texttt{m}'$, $\cdots$, and for local variables, ranged over by x, y, $\cdots$. A possible empty sequence of names or syntactic categories of the following grammar is written by over-lining the name or the syntactic category, respectively. For instance a

[3] Actually, $\texttt{JVML}_d$ has a minor difference with respect to `JVML`: in `JVML`, local variables are addressed by non-negative integers instead of names.

sequence of local variables is written $\overline{x}$. Class files CF are defined by the grammar:

$$\begin{array}{lll@{\qquad}lll} CF & ::= & \mathit{class}\ \mathtt{C}\ \{\mathit{fields} : \overline{FD}\ \ \mathit{methods} : \overline{MD}\} & MD & ::= & \mathtt{T}\ \mathtt{m}\ (\mathtt{C}, \overline{\mathtt{T}})\ P\ P \\ FD & ::= & \mathtt{C.f} : \mathtt{T} & \mathtt{T} & ::= & \top \mid \mathtt{int} \mid \mathtt{C} \end{array}$$

where $\top$ is a special type that include all the other types (any value of any type has also type $\top$). This type will represent values that are unusable in our static semantics. The type name `C` represents a class type, *which is never recursive* in $\mathtt{JVML}_d$.

Instructions *Instr* of $\mathtt{JVML}_d$ bytecode are of the following form:

$$\begin{array}{lll} \mathit{Instr} ::= & & \mathtt{inc} \mid \mathtt{pop} \mid \mathtt{push} \mid \mathtt{load}\ x \mid \mathtt{store}\ x \mid \mathtt{if}\ L \mid \mathtt{goto}\, L \\ & \mid & \mathtt{new}\,\mathtt{C} \mid \mathtt{putfield}\,\mathtt{C.f} : \mathtt{T} \mid \mathtt{getfield}\,\mathtt{C.f} : \mathtt{T} \mid \mathtt{monitorenter} \mid \mathtt{monitorexit} \\ & \mid & \mathtt{invokevirtual}\ \mathtt{C.m}(\overline{\mathtt{T}}) \mid \mathtt{start}\ \mathtt{C} \mid \mathtt{return} \end{array}$$

The informal meaning of these instructions is as follows:

- `inc` increments the content of the stack; `pop` and `push`, respectively, pops and pushes the integer 0 on the stack; `load` x and `store` x respectively loads the value of x on the stack and pops the top value of the stack by storing it in x; `if` L pops the top value of the stack and either jumps to the instruction at address L, if it is nonzero, or goes to the next instruction; `goto` L is the unconditional jump;
- `new C` allocates a new object of type `C`, initializes it and pushes it on top of the stack; `putfield C.f : T` pops the value on the stack and the underlying object value, and assigns the former to the field `f` of the latter; `getfield C.f : T` pops the object on the stack and pushes the value in the field `f` of that object;
- `monitorenter`, `monitorexit` are the synchronization primitives that pop the object on the stack and respectively lock and unlock it;
- `invokevirtual` $\mathtt{C.m}(\mathtt{T}_1, \cdots, \mathtt{T}_n)$ pops n values from the stack (the arguments of the invocation) and dispatches the method `m` on the object on top of the stack; when the method terminates, the returned value is pushed on the stack;
- `start C` creates and starts a new thread for the object on top of the stack. This operation corresponds to `invokevirtual java/lang/Thread/start()` on a thread of class `C` in `JVML`. We separate it from `invokevirtual` in order to provide more structure to our semantics (because it has an effect on the set of threads – see the operational semantics in the Appendix, where we also consider the instruction `join`);
- `return` terminates program execution.

The bytecode in Figure 1 is written in a sugared extension of $\mathtt{JVML}_d$. In particular, `aload` and `iload` correspond to our `load` instruction (when the argument is an object or an integer, respectively), `dup` duplicates the top of the stack, and `invokespecial` is the method invocation of the constructor of the class.

4 The static semantics

In this section we define the typing rules that are at the base of the analysis technique. In order to simplify the presentation and the notation, we decided to drop the effect analysis, which verifies that threads do not access to common objects in inconsistent ways (for example, one reading and the other one writing). In particular, we assume that fields are read-only as they cannot be modified after the initialisation (which is done by constructors that, in turn, are not concurrent). For reviewers' sake, the Appendix reports the complete analysis.

Type values, flattened record types, and record types.

The static semantics traces dependencies between objects by using *symbolic names*. We will use a set of *object names*, ranged over by a, b, $\cdots$ that includes *void* and the *thread names* (threads are objects in `Java`); when a name is a thread name, we use t, t', $\cdots$. We also use x, y, z, $\cdots$ to range over (generic) *names* and X, Y, Z, $\cdots$ to range over *variable names* to be used in the typing of methods and to be instantiated when methods are invoked.

Let *type values* τ and *flattened record types* ϕ be the terms

$$\tau \quad ::= \quad \top \mid \mathtt{int} \mid X \mid a \qquad\qquad \phi \quad ::= \quad ([\overline{\mathtt{f}:\tau}], \mathtt{C})$$

It is worth to remark that flattened record types also bear their class set, which is a singleton in $\mathtt{JVML}_d$ (but not in full `Java`, where sets may contain several classes because of inheritance). Types τ and ϕ are *unstructured* because fields are names, i.e. they are pointers to records whenever the corresponding type is a class. This expedient allows us to analyse *aliasing* in a simple way. However, the type system also uses *structured types* (e.g in method types). Let (*structured*) *record types* ρ be the terms (this definition is correct because $\mathtt{JVML}_d$ class types are *not recursive*):

$$\rho \quad ::= \quad \top \mid \mathtt{int} \mid X \mid (a[\overline{\mathtt{f}:\rho}], \mathtt{C}) \ .$$

We shorten $a[\,]$ into a; henceforth *void*$[\,]$ is shortened into *void*. Let $\mathit{root}(\cdot)$ be the function defined as follows: $\mathit{root}(\mathit{void}) = \varepsilon$, $\mathit{root}(a[\overline{\mathtt{f}:\rho}]) = a$, $\mathit{root}(\cdot)$ is the identity otherwise.

Environments.

The type rules use *environments* Γ that map names to type values or to flattened record types. There are two basic operations on environments: one for *sequential composition* – the *update* $\Gamma[\Gamma']$ – and one for *parallel composition* – the *merge* $\Gamma + \Gamma'$. They are defined as follows

$$\Gamma[\Gamma'](a) \ = \begin{cases} \Gamma'(a) & \text{if } a \in \mathit{dom}(\Gamma') \\ \Gamma(a) & \text{otherwise} \end{cases} \qquad (\Gamma+\Gamma')(a) \ = \begin{cases} \Gamma(a) & \text{if } a \in \mathit{dom}(\Gamma) \backslash \mathit{dom}(\Gamma') \\ \Gamma'(a) & \text{if } a \in \mathit{dom}(\Gamma') \backslash \mathit{dom}(\Gamma) \\ \Gamma(a) & \text{if } \Gamma(a) = \Gamma'(a) \end{cases}$$

For example, let $\Gamma = a \mapsto ([\mathtt{f}:b], \mathtt{C})$ and $\Gamma' = a \mapsto ([\mathtt{f}:c], \mathtt{C})$. Then $\Gamma[\Gamma'] = a \mapsto ([\mathtt{f}:c, \mathtt{C})$, which is the standard update of an environment (this is used in constructors), while $\Gamma + \Gamma'$ is undefined (because there is a *race condition* on the field `f` of a).

There is a straightforward way to get an environment from a record type and, conversely, to transform an environment and an object name into its (structured) record type. This way is defined by $\mathit{env}(\cdot)$ and $\mathit{mk\_tree}(\cdot)$ below:

$$\mathit{env}(\rho) \ = \begin{cases} \varnothing & \text{if } \rho \in \{\top, \mathtt{int}, X\} \\ a \mapsto ([\overline{\mathtt{f}:\mathit{root}(\rho)}], \mathtt{C}) + (+_{\rho' \in \overline{\rho}}\, \mathit{env}(\rho')) & \text{if } \rho = (a[\overline{\mathtt{f}:\rho}], \mathtt{C}) \end{cases}$$

$$\mathit{mk\_tree}(\Gamma, \tau) \ = \begin{cases} \tau & \text{if } \tau \in \{\top, \mathtt{int}, X\} \\ (a[\overline{\mathtt{f}:\mathit{mk\_tree}(\Gamma,\tau')}], \mathtt{C}) & \text{if } \tau = a \text{ and } \Gamma(a) = ([\overline{\mathtt{f}:\tau'}], \mathtt{C}) \end{cases}$$

(notice that $\mathit{env}(\cdot)$ is partial and it and $\mathit{mk\_tree}(\cdot)$ are well defined as long as class types are *not recursive*, as it is indeed the case for $\mathtt{JVML}_d$). We finally define $\mathit{typeof}(\Gamma, \tau) = \mathtt{C}$ if $\Gamma(\tau) = ([\overline{\mathtt{f}:\tau'}], \mathtt{C})$; $\mathit{typeof}(\Gamma, \tau) = \mathtt{int}$ if $\tau = \mathtt{int}$; undefined otherwise. Similarly, for record types ρ.

In the following we will use sets of record types, noted $\mathscr{T}, \mathscr{R}, \cdots$.

```
Main(this | t,u) =   Object.init(x | t,u):x[] + Object.init(y | t,u):y[]
                     + buildNetwork(this,_,x,x | t,u) + buildNetwork(this,_,x,y | t,u)

takeLocks(this,x,y | t,u) = t:(u,x) & t:(x,y)

buildNetwork(this,_,x,y | t,u) =   takeLocks(this,x,y | t,u) + Object.init(z | t,u):z[]
                               + Network$1.init(t1,this,x,z | t, z):t1[this$0:this[], val$x:x[], val$z: z[]]
                               + Network$1.run(t1 | t1,u1)
                               + Network$1.run(t1 | t1,u1) & buildNetwork(this,_,z,y | t,u)

Object.init(this | t, u):this[] =   0

Network$1.init(this, x1, x2, x3 | t, u):this[this$0:x1, val$x:x2, val$z:x3] =   0

Network$1.run(this[this$0:x1, val$x:x2, val$z:x3] | t, u) =   takeLocks(x1, x2, x3 | t, u)
```

Figure 2 `Network`'s lams

Lams.

Behavioural types are *lams* [10], noted ℓ, whose syntax is

$$\ell \ ::= \ \ \mathtt{0} \ \ | \ \ (a,b)_t \ \ | \ \ \mathtt{C.m}(\overline{\rho}) \to \rho' \ \ | \ \ (\nu\, a)\ell \ \ | \ \ \ell \,\&\, \ell \ \ | \ \ \ell + \ell$$

The type $\mathtt{0}$ is the empty type; $(a,b)_t$ specifies a dependency between the object a and the object b that has been created by the thread t. The term $\mathtt{C.m}(\overline{\rho}) \to \rho'$ defines the invocation of $\mathtt{C.m}$ with arguments $\overline{\rho}$ and with returned record type ρ'. The argument sequence $\overline{\rho}$ has always at least three elements in our case; in particular the first element is the record type of the carrier, while the last two elements of the tuple $\overline{\rho}$ are respectively the thread t that performed the invocation and the last object name whose lock has been acquired by t. These two informations are used by the analyzser to build the right dependencies between callers and callees. The operation $(\nu\, a)\ell$ creates a new name a whose scope is the type ℓ; the operations $\ell \,\&\, \ell'$ and $\ell + \ell'$ define the conjunction and disjunction of the dependencies in ℓ and ℓ', respectively. The operators $+$ and $\&$ are associative and commutative. We shorten $\ell_1 + \cdots + \ell_n$ and $\ell_1 \& \cdots \& \ell_n$ into $\sum_{i \in 1..n} \ell_i$ and $\&_{i \in 1..n} \ell_i$, respectively.

A *lam program* is a pair $\big(\mathscr{L}, \ell\big)$, where $\mathscr{L}$ is a *finite set* of *function definitions*

$$\mathtt{C.m}(\overline{\rho}) \to \rho' \ = \ \ell_{\mathtt{C.m}}$$

with $\ell_{\mathtt{C.m}}$ being the *body* of $\mathtt{C.m}$, and ℓ is the *main lam*. We notice that the type ρ' is considered an argument of the lam function as well.

The lams of the `Network`'s code in Figure 1 are shown in Figure 2 (lams have been simplified for easing the readability).

Behavioural Class Table.

A *behavioural class table* BCT is a map from pairs $\mathtt{C.m}$ to *method types* $(\bar{\rho}, t, a) \to (\nu\, \overline{a'})\langle \rho, \mathscr{T}, \mathscr{R}, \overline{\rho'}, \ell \rangle$ where

- $\bar{\rho}$ is the tuple of record types of the carrier and of the arguments; t is the thread name of the caller and a is the last lock taken (and not released) by the caller;
- $(\nu\, \overline{a'})$ are the names that have been created by the method (and occur in the part in angular brackets);
- ρ is the record type of the returned value; this type may contain fresh object names that are created by the method body;
- $\mathscr{T}$ is the set of threads that have been created by the method (this set only contains record types whose root names are fresh);

- $\mathscr{R}$ is similar to $\mathscr{T}$. Because of recursion, `C.m` may create infinitely many threads; in this case $\mathscr{R}$ is a finite representation of an infinite set;
- $\overline{\rho'}$ is the record type highlighting the changes to the arguments performed by the invocation (with the restriction we have, this is relevant for constructors);
- ℓ is the lam of the method `C.m`.

Let σ be a substitution that renames object names and replaces variable names with record types and such that there is no clash of names $\overline{c}$ with names in σ. The *instance of a method type* $(\overline{\rho}, t, a) \to (\nu\, \overline{c})\langle \rho, \mathscr{T}, \mathscr{R}, \overline{\rho'}, \ell\rangle$ is

$$(\overline{\rho}, t, a)\sigma \to (\nu\, \overline{c})(\langle \rho, \mathscr{T}, \mathscr{R}, \overline{\rho'}, \ell\rangle\sigma)$$

Let $(\overline{\rho}, t, a) \to (\nu\, \overline{c})\langle \rho, \mathscr{T}, \mathscr{R}, \overline{\rho'}, \ell\rangle$ be an instance of $\textsc{bct}(\texttt{C.m})$. When we write $\textsc{bct}(\texttt{C.m})(\overline{\rho}, t, a)(\overline{b})$ we mean the tuple $\langle \rho, \mathscr{T}, \mathscr{R}, \overline{\rho'}, \ell\rangle\{^{\overline{b}}/_{\overline{c}}\}$.

Typing judgments.

The judgment of $\texttt{JVML}_d$ bytecode P will be $\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : \ell\,;\, \Psi$ where:

- $\textsc{bct}$ is the behavioural class table; $\Gamma, F, S, Z, \mathscr{T}, \mathscr{R}$ are *vectors* indexed by the addresses in $dom(P)$;
- the environment Γ_i is the environment at address i;
- the map F_i maps local variables to type values – we let $F_\top$ be the map such that $F(x) = \top$, for every x;
- S_i is a sequence of type values;
- Z_i is the *sequence of object names* locked at address i;
- $\mathscr{T}_i$ and $\mathscr{R}_i$ are similar to the corresponding sets in method types;
- t is the symbolic thread name that executes P;
- ℓ is *part of* the behavioural type of P at address i. The full behavioural type also includes Z_i, $\mathscr{T}_i$, and $\mathscr{R}_i$ (see below);
- Ψ may be either empty, in this case the judgment is shortened into $\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : \ell$, or a tuple $(\rho, Z_i, \mathscr{T}_i, \mathscr{R}_i, \Gamma_i)$. Let $\bigsqcup$ be the commutative and associative operator defined as follows

$$\begin{array}{rcl} (\rho, Z_i, \mathscr{T}_i, \mathscr{R}_i, \Gamma_i) \bigsqcup \varnothing & \stackrel{def}{=} & (\rho, Z_i, \mathscr{T}_i, \mathscr{R}_i, \Gamma_i) \\ (\rho, Z_i, \mathscr{T}_i, \mathscr{R}_i, \Gamma_i) \bigsqcup (\rho', Z_i, \mathscr{T}_j, \mathscr{R}_j, \Gamma_j) & \stackrel{def}{=} & (\rho, Z_i, \mathscr{T}_i \cup \mathscr{T}_j, \mathscr{R}_i \cup \mathscr{R}_j, \Gamma_i + \Gamma_j\,) \end{array}$$

($\bigsqcup$ is only defined on tuples with the elements ρ and Z_i equal).

Let $Z_i = a_1 \cdots a_n$ be the sequence of (locked) object names at instruction i; we assume that the leftmost object name is the more recent one that has been taken. The functions $\lceil Z_i \rceil$ and $\widehat{Z_i}^t$ are defined as follows

$$\lceil Z_i \rceil \;=\; a_1 \qquad\qquad \widehat{Z_i}^t \;=\; \&_{j\in 2..n}(a_j, a_{j-1})_t$$

We observe that Z_i may contain twice the same object name; this means that the thread has acquired twice the corresponding lock. For instance $Z_i = a \cdot a$. In this case $\widehat{Z_i}^t = (a, a)_t$, which is not a circular dependency – this is the reason why we index dependencies with thread names.

Let $lock_t$ be a special object name that is paired with the thread t. The lam $\widehat{\mathscr{T}_i}$ is the term

$$\widehat{\mathscr{T}_i} \;=\; \mathop{\&}_{\rho\in\mathscr{T}_i} typeof(\rho).\texttt{run}(\rho, \texttt{C}), t, lock_t)$$

Stack manipulation instructions

$$\frac{\begin{array}{c} P[i] = \texttt{inc} \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_i = \Gamma_{i+1} \quad F_i = F_{i+1} \quad S_i = \texttt{int} \cdot S' = S_{i+1} \\ Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

$$\frac{\begin{array}{c} P[i] = \texttt{pop} \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = \tau \cdot S_{i+1} \\ Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0} \qquad \frac{\begin{array}{c} P[i] = \texttt{push} \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_i = \Gamma_{i+1} \quad F_i = F_{i+1} \quad \texttt{int} \cdot S_i = S_{i+1} \\ Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

$$\frac{\begin{array}{c} P[i] = \texttt{load}\ x \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_i = \Gamma_{i+1} \quad F_i = F_{i+1} \quad S_{i+1} = F_i(x) \cdot S_i \\ Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0} \qquad \frac{\begin{array}{c} P[i] = \texttt{store}\ x \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_i = \Gamma_{i+1} \quad F_{i+1} = F_i[x \mapsto \tau] \quad S_i = \tau \cdot S_{i+1} \\ Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

Control flow instructions

$$\frac{\begin{array}{c} P[i] = \texttt{goto}\ L \qquad L \in \mathit{dom}(P) \\ \Gamma_i = \Gamma_L \quad F_i = F_L \quad S_i = S_L \\ Z_i = Z_L \quad \mathscr{T}_i = \mathscr{T}_L \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0} \qquad \frac{\begin{array}{c} P[i] = \texttt{if}\ L \qquad i+1, L \in \mathit{dom}(P) \\ \Gamma_{i+1} = \Gamma_i = \Gamma_L \quad F_i = F_{i+1} = F_L \\ S_i = \texttt{int} \cdot S_{i+1} \quad S_{i+1} = S_L \quad Z_i = Z_{i+1} = Z_L \\ \mathscr{T}_i = \mathscr{T}_{i+1} = \mathscr{T}_L \quad \mathscr{R}_i = \mathscr{R}_{i+1} = \mathscr{R}_L \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

Return

$$\frac{P[i] = \texttt{return} \qquad S_i = \tau \cdot S'}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0\ ;\ (\mathit{mk\_tree}(\Gamma_i, \tau), Z_i, \mathscr{T}_i, \mathscr{R}_i, \Gamma_i)}$$

Figure 3 Type rules for $\texttt{JVML}_d$ programs – Part I

where `run` is the method that is invoked by the `Java` instruction t`.start()`. We notice that `start` has no arguments, except the callee t: the parameters are passed to the methods through the fields of t.

The lam $\widetilde{\mathscr{R}_i}$ is the term

$$\begin{aligned} \widetilde{\mathscr{R}_i} &= \&_{\rho \in \mathscr{R}_i} \texttt{RUN}(\rho) \\ \texttt{RUN}((t[\texttt{f} : \rho], \texttt{C})) &= \texttt{C.run}((t[\texttt{f} : \rho], \texttt{C}), t, \mathit{lock}_t)\ \&\ (\nu\, t')\, \texttt{RUN}((t'[\texttt{f} : \rho], \texttt{C})) \end{aligned}$$

The presence of recursion in $\widetilde{\mathscr{R}_i}$ is the main difference with $\widehat{\mathscr{T}_i}$. In fact, $\mathscr{R}_i$ collects threads that have been created by recursive or iterative methods. This means that there may be several threads with equal fields but different root names running in parallel and that may compete for the same locks.

Whenever $i \in \mathit{dom}(P)$, we write "i is recursive" if i is within a recursive or mutually recursive method. Otherwise we write "i is not recursive".

Type rules.

The type rules for a $\texttt{JVML}_d$ programs are reported in Figures 3 and 4.

The function names(i). This function takes an address i and returns a tuple of names whose length depends on the address. It is the technical expedient to keep the set $\mathscr{R}$ finite when methods are either recursive or iterative. For example, let `C.m` be a recursive method and let $(\overline{\rho}, t, a) \to (\nu\, \overline{a'})\langle \rho, \mathscr{T}, \mathscr{R}, \overline{\rho'}, \ell \rangle$ be its type. If `C.m` starts a new thread t and, in a successive instruction at address i, invokes itself recursively (this happens for instance in method `buildNetwork` of Figure 1) then the set $\mathscr{R}_{i+1}$ will be equal to $\mathscr{R} \cup \{t\}$. If, in addition,

Method invocations

$$\frac{\begin{array}{c} P[i] = \texttt{invokevirtual}\ \ \texttt{C.m}\ (\texttt{T}_0, \cdots, \texttt{T}_k) \qquad i+1 \in \mathit{dom}(P) \qquad S_i = \tau_k \cdots\ \tau_0 \cdot S' \\ \mathit{typeof}(\Gamma_i, \tau_0) = \texttt{C} \quad \mathit{mk\_tree}(\Gamma_i, (\tau_0, \cdots, \tau_k)) = (\rho_0, \cdots, \rho_k) \\ \overline{b} = \mathit{names}(i) \quad \overline{b} \cap \mathit{var}(\rho_0, \cdots, \rho_k, t, \lceil Z_i \rceil) = \varnothing \\ \textsc{bct}(\texttt{C.m})(\rho_0, \cdots, \rho_k, t, \lceil Z_i \rceil)(\overline{b}) = \langle \rho, \mathscr{T}', \mathscr{R}', (\rho'_0, \cdots, \rho'_k), \ell \rangle \\ \Gamma_{i+1} = \Gamma_i[\mathit{env}(\rho) + (\mathop{+}_{i\in 0..k} \mathit{env}(\rho'_i))] \\ S_{i+1} = \mathit{root}(\rho) \cdot S' \quad F_{i+1} = F_i \quad Z_{i+1} = Z_i \\ \mathscr{T}_{i+1}, \mathscr{R}_{i+1} = \begin{cases} \mathscr{T}_i \cup \mathscr{T}',\ \mathscr{R}_i \cup \mathscr{R}' & \text{if } i \text{ is not recursive and } (\mathscr{T}_i \cup \mathscr{R}_i) \cap (\mathscr{T}' \cup \mathscr{R}') = \varnothing \\ \mathscr{T}_i,\ \mathscr{R}_i \cup \mathscr{R}' \cup \mathscr{T}' & \text{otherwise} \end{cases} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : \texttt{C.m}(\rho_0, \cdots, \rho_k, t, \lceil Z_i \rceil) \to \rho}$$

$$\frac{\begin{array}{c} P[i] = \texttt{start C} \qquad i+1 \in \mathit{dom}(P) \qquad S_i = t' \cdot S_{i+1} \\ \mathit{typeof}(\Gamma_i, t') = \texttt{C} \quad \mathit{mk\_tree}(\Gamma_i, t') = \rho \quad \overline{b} = \mathit{names}(i) \quad \overline{b} \cap \mathit{var}(\rho, t', \mathit{lock}_{t'}) = \varnothing \\ \textsc{bct}(\texttt{C.run})(\rho, t', \mathit{lock}_{t'})(\overline{b}) = \langle \mathit{void}, \mathscr{T}', \mathscr{R}', \rho', \ell \rangle \\ \Gamma_{i+1} = \Gamma_i + \mathit{env}(\rho') \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \\ \mathscr{T}_{i+1}, \mathscr{R}_{i+1} = \begin{cases} \mathscr{T}_i \cup \{\tau\} \cup \mathscr{T}',\ \mathscr{R}_i \cup \mathscr{R}' & \text{if } i \text{ is not recursive and } \tau \notin \mathscr{T}_i \cup \mathscr{R}_i \\ \mathscr{T}_i,\ \mathscr{R}_i \cup \mathscr{R}' \cup \mathscr{T}' \cup \{\tau\} & \text{otherwise} \end{cases} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

Locking instructions

$$\frac{\begin{array}{c} P[i] = \texttt{monitorenter} \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \\ Z_{i+1} = a \cdot Z_i \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0} \qquad \frac{\begin{array}{c} P[i] = \texttt{monitorexit} \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \\ Z_{i+1} = Z_i \backslash a \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

Object manipulation instructions

$$\frac{\begin{array}{c} P[i] = \texttt{new C} \qquad i+1 \in \mathit{dom}(P) \quad a = \mathit{names}(i) \quad \mathsf{fields}(\texttt{C}) = \bar{\texttt{f}} \\ \Gamma_{i+1} = \Gamma_i[a \mapsto ([\overline{\texttt{f} : \top}], \texttt{C})] \\ F_i = F_{i+1} \quad S_{i+1} = a \cdot S_i \quad Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

$$\frac{\begin{array}{c} P[i] = \texttt{putfield C.f : T} \qquad i+1 \in \mathit{dom}(P) \\ S_i = \tau \cdot a \cdot S_{i+1} \quad \Gamma_i(a) = ([\texttt{f} : \tau', \overline{\texttt{f} : \tau'}], \texttt{C}) \\ \Gamma_{i+1} = \Gamma_i[a \mapsto ([\texttt{f} : \tau, \overline{\texttt{f} : \tau'}], \texttt{C})] \quad F_i = F_{i+1} \\ Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0} \qquad \frac{\begin{array}{c} P[i] = \texttt{getfield C.f : T} \qquad i+1 \in \mathit{dom}(P) \\ S_i = a \cdot S' \quad \Gamma_i(a) = ([\texttt{f} : \tau, \overline{\texttt{f} : \tau}], \texttt{C}) \\ \Gamma_{i+1} = \Gamma_i[a \mapsto ([\texttt{f} : \tau, \overline{\texttt{f} : \tau}], \texttt{C})] \\ S_{i+1} = \tau \cdot S' \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \\ \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : 0}$$

Method definitions

$$\frac{\begin{array}{c} \textsc{bct}(\texttt{C.m}) = (\rho_0, \cdots, \rho_k, t, a) \to (\nu \overline{a}')\langle \rho, \mathscr{T}', \mathscr{R}', \overline{\rho'}, \ell \rangle \quad F_1 = F_\top[0 \mapsto \mathit{root}(\rho_0), \cdots, k \mapsto \mathit{root}(\rho_k)] \\ \Gamma_1 = \mathop{+}_{i\in 0..k} \mathit{env}(\rho_i) \quad S_1 = \varepsilon \quad Z_1 = a \quad \mathscr{T}_1 = \varnothing \quad \mathscr{R}_1 = \varnothing \\ \Big(\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, i \vdash_t P : \ell_i\ ;\ \Psi_i\Big)^{i \in \mathit{dom}(P)} \qquad \ell = \sum_{i\in \mathit{dom}(P)} \widehat{\mathscr{T}_i}\ \&\ \widecheck{\mathscr{R}_i}\ \&\ \widehat{Z_i}^{t}\ \&\ \ell_i \\ \bigsqcup_{i \in \mathit{dom}(P)} \Psi_i = (\vartheta, a, \mathscr{T}', \mathscr{R}', \Gamma') \\ \overline{\rho'} = \mathit{mk\_tree}(\Gamma', (\mathit{root}(\rho_0), \cdots, \mathit{root}(\rho_k))) \quad \overline{a'} = \mathit{fn}(\rho, \mathscr{T}', \mathscr{R}', \overline{\rho'}) \backslash\ \mathit{fn}(\rho_0, \cdots, \rho_k, t, a) \end{array}}{\textsc{bct} \vdash \texttt{T C.m}\ (\texttt{T}_1, \cdots, \texttt{T}_k)\ P}$$

Figure 4 Type rules for $\texttt{JVML}_d$ programs – Part II

the instruction at $i+1$ is `return`, we obtain the equation $\mathscr{R} = \mathscr{R} \cup \{t\}$. This equation has no solution if t is always a fresh name (as it is the case in the `JVML` semantics); on the contrary it has solution $\mathscr{R} = \{t\}$ if $names(i) = t$. This second alternative has been our choice. In turn, this choice has an important impact on the analysis. In particular, we decided to constrain the threads created by recursive invocations or iterative codes to have the *same tree structure* – i.e. the corresponding nodes are equal. The analysis of behavioural types – the set $\mathscr{R}$ is part of them – will take care of this over-approximation. Actually, the evaluation of $\mathscr{R}$ spawns infinitely many lam functions with arguments bearing pairwise different fresh root names (*cf.* the function $\widecheck{\mathscr{R}}$), while it is not the case for the type of $\mathscr{T}$.

Comments about the type rules. The type rules that are important for the deadlock analysis are collected in Figure 4 and we will focus on them; the rules in Figure 3 are almost standard, see for instance [15].

The rule for `invokevirtual` computes the instance of $\text{BCT}(\texttt{C.m})$ according to the arguments of the invocation. Let $\langle \rho, \mathscr{T}', \mathscr{R}', \overline{\rho'}, \ell \rangle$ be such instance. Since `invokevirtual` is a sequential invocation, its effects are reported in the environment Γ_{i+1} that types the next instruction. There are three effects: (*i*) the returned value, because it may be an object created by the method; (*ii*) the updates of the arguments of the invocation – the tuple $\overline{\rho'}$ – that, according to the restrictions we have, happens when `C.m` is a constructor; (*iii*) the threads that have been spawned by `C.m` – the set $\mathscr{T}'$, in case `C.m` is not recursive, and the set $\mathscr{R}'$, in case `C.m` is recursive or invokes methods that recursively spawn new threads. The sets $\mathscr{T}_{i+1}$ and $\mathscr{R}_{i+1}$ contain the threads that have been created by P up-to i. It is important to keep $\mathscr{R}_{i+1}$ as small as possible because this impacts on the precision of our analysis: we are precise on threads in $\mathscr{T}_{i+1}$ (because they are created exactly once) while we are over-approximate on threads in $\mathscr{R}_{i+1}$. So,

- if `C.m` is not recursive and *the instruction i is not inside an iteration* (that is, it is executed once) then $\mathscr{T}_{i+1}$ and $\mathscr{R}_{i+1}$ are $\mathscr{T}_i \cup \mathscr{T}'$ and $\mathscr{R}_i \cup \mathscr{R}'$, respectively. The foregoing sentence in italics is expressed by the constraint $(\mathscr{T}_i \cup \mathscr{R}_i) \cap (\mathscr{T}' \cup \mathscr{R}') \neq \varnothing$. We explain this constraint with an example. Assume that $i+1$ is a `goto` i (this means that the invocation is performed infinitely many times). Then, by the type rule for `goto`, $\mathscr{T}_i = \mathscr{T}_{i+1}$ and $\mathscr{R}_i = \mathscr{R}_{i+1}$. Therefore the threads in $\mathscr{T}' \cup \mathscr{R}'$ must be already in $\mathscr{T}_i \cup \mathscr{R}_i$.
- otherwise both $\mathscr{T}'$ and $\mathscr{R}'$ are added to $\mathscr{R}_{i+1}$.

The lam in the conclusion contains the invocation to `C.m`. We observe that this invocation is part of the behavioural type of the instruction: the full type also includes the terms $\widehat{\mathscr{T}_i}$ & $\widecheck{\mathscr{R}_i}$ and the dependencies of the sequence of locks in Z_i – the term $\widehat{Z_i}^t$ – see the rule for method invocations.

The rule for `start` computes the instance of $\text{BCT}(\texttt{C.run})$ where `C` is the type of the thread t' on top of the stack S_i. In our case $\rho' = \rho$ because fields cannot be updated. The environment Γ_{i+1} contains the threads spawned by `C.run` – the sets $\mathscr{T}'$ and $\mathscr{R}'$ in the premise. The sets $\mathscr{T}_{i+1}$ and $\mathscr{R}_{i+1}$ are augmented with the thread that has been just spawned and those spawned by it as done with the rule for `invokevirtual`.

The rules for the locking instructions `monitorenter` and `monitorexit` update the map Z as expected. We notice that this update has an effect on the lam $\widehat{Z_{i+1}}^t$ by augmenting or reducing the dependencies, respectively.

The rule for method declarations `C.m` verifies whether what is declared in $\text{BCT}(\texttt{C.m})$ does match with what is checked by the judgments of the instructions in its body. Assuming that t is the name of the current thread and a is the last lock that has been acquired, then we constrain the first instruction of `C.m` to be typed with F_1 such that the first k local variables are set to the arguments of the invocation and Γ_1 defining such arguments. As regards S_1, $\mathscr{T}_1$ and $\mathscr{R}_1$, they are all empty, while $Z_1 = a$. The matching between BCT(`C.m`) and what is checked by the judgments is performed by collecting the tuples $(\vartheta_i, Z_i, \mathscr{T}_i, \mathscr{R}_i, \Gamma_i)$ of the `return` statements and merging the results. Notice that, if `C.m` does not perform any concurrent operation (`start`, `monitorenter`, `monitorexit`) and does not perform method invocations then every $\mathscr{T}_i$ is empty and every Z_i is a. Therefore, the lam of every instruction is always `0`.

5 The analysis of circularities in lams

Once behavioural types have been computed for the whole $\texttt{JVML}_d$ program, we can analyse the type of the *main* method. The analysis uses an extension of the algorithm defined in [10, 14] that we briefly overview in this section. The paper [9] reports the pseudo-code of the algorithm that is implemented in our verifier; here we give an informal presentation.

The semantics of lams is very simple: it amounts to unfolding method invocations. The critical points are that (i) every invocation may create new fresh names and (ii) the method definitions may be recursive. These two points imply that a lam model may have infinite states, which makes any analysis nontrivial. It is worth to recall that the states of lams are conjunctions ($\&$) of dependencies and method invocation (because types with disjunctions $+$ are modelled by sets of states with conjunctive dependencies).

The results of [10, 14] allow us to reduce the analysis to *finite* models, *i.e.* finite disjunctions of finite conjunctions of dependencies. In turn, this finiteness makes possible to decide the presence of a so-called *circularities*, namely terms such as $(a, b)_t \& (b, a)_{t'}$.

The difference with [10, 14] is that there the dependencies are not indexed by thread names: here we use more informative dependencies in order to cope with `Java` reentrant locks. In particular $(a, b)_t \& (b, a)_t$ is not a circularity and, when $t \neq t'$, we carefully separate it from $(a, b)_t \& (b, a)_{t'}$. The definition of *transitive closure*, which is the base of the notion of circularity, is therefore new. Let $t \neq t'$ and let $\checkmark$ be a special object name. Let also ℓ be a conjunction $\&$ of dependencies.

- The *transitive closure* of ℓ, noted ℓ^+, is the least conjunction that contains ℓ and such that if $(a, b)_t \& (b, c)_{t'}$ is a subterm of ℓ^+ then either (i) $(a, c)_\checkmark$ is a subterm of ℓ^+, if $t \neq t'$, or (ii) $(a, c)_t$ is a subterm of ℓ^+, if $t = t'$.
- We say that ℓ has a *circularity* if there is a such that $(a, a)_\checkmark$ is a subterm of ℓ^+.

For example $\ell = (a, b)_t \& (b, a)_t \& (b, c)_{t'}$ has no circularity because $\ell^+ = (a, b)_t \& (b, a)_t \& (a, a)_t \& (b, b)_t \& (b, c)_{t'} \& (a, c)_\checkmark$ does not contain any pair $(a, a)_\checkmark$ (this symbol $\checkmark$ is a special thread name indicating that the dependency is due to the contributions of two or more threads).

▸ Remark. Actually, our verifier uses a notion of transitivity that is a refinement of the one described above. In particular, dependencies are labelled by the sequence of threads that contribute to them. These more informative labels allows us to compute transitive closures of terms like $(a, b)_t \& (b, c)_t \& (a, c)_{t'} \& (c, b)_{t'}$ in a more precise way. In fact, in this case, the mutual exclusion on the initial object a makes the concurrent presence of $(b, c)_t$ and $(c, b)_{t'}$ not possible. We finally notice that, using sequences of thread names as indices of dependencies allows us to reconstruct, at least in part, the execution path that produces the error. This is crucial for detecting false positives.

6 Intermezzo: correctness results

The proof of correctness of our type system is long, even if almost standard. In this section we overview it by highlighting the main parts; the details of this proof can be found in the Appendix. The first part of the proof addresses the soundness of the type system in Section 4. This requires

1. an operational semantics of $\texttt{JVML}_d$;
2. an extension of the type system to handle `JVM` runtime configurations.

Then, as usual with type systems, the soundness is represented by a subject reduction theorem expressing that, if a `JVM` configuration cn has lam ℓ and cn reduces to a configuration cn' then (i) cn' is also well-typed and (ii) if ℓ' is the type of cn' then ℓ and ℓ' are in a relation called *later stage* and noted $\ell \succcurlyeq \ell'$.

The second part of the proof is about the correctness of the later stage relation with respect to the deadlock analysis. We first demonstrate that the lam of a deadlocked configuration always contains a *circularity* between names. Then we demonstrate that, if $\ell \succcurlyeq \ell'$ and ℓ *has no circularity* then also ℓ' has no circular dependency.

As a byproduct of the above results we get that, if the lam of a $\texttt{JVML}_d$ program has no circularity then the $\texttt{JVML}_d$ program is deadlock-free.

7 Related Work and `JaDA`

Several techniques have been designed for detecting deadlocks of `Java` programs. We discuss separately those using static-time techniques and those requiring either model generation or runtime executions.

Static approaches

Type Systems. Few works use type systems for the deadlock detection in `Java`. To our knowledge, the most complete work is [4], which defines a type system that derives the order of lock acquisitions in *SafeJava* programs (a subset of `Java` *with annotations* written as part of the code). Well-typed programs will be verified deadlock free. An extension of this type system for `JVML` has been defined in [18]. Our technique differs from this approach in two key aspects. First of all, we aim at a fully automatic tool allowing to analyse existing programs without user intervention (or with a poor intervention that providing behavioural types of native methods because their code is not available). In addition, our types are behavioural; therefore deadlock-freedom is not a property of the type system, but it is verified by a solver that evaluates behavioural types.

Data Flow Analysis. A standard approach for detecting circular dependencies in concurrent programs is based on the construction of an execution flow graph and the search for cycles within this graph. In order to cope with aliasing (processes may use different names to access to the same resource), the definition of the execution flow graph uses a data-flow analysis technique, as in `Jlint` [1]. In [20], data flow analysis is used for defining a formal set of rules that specify the lock order (the mechanism is similar to one used in the type system of this paper). As in our case, [20] has a sound strategy for detecting re-entrant locks; however the re-entrance in their case is restricted to lock expressions that only use local variables (it is not possible to use fields). Finally, the technique in [20] does not detect circularities with a common prefix (see Remark in Section 5), thus leading to a higher number of false positive outcomes. Data flow analysis is also used by Nayik [16] for verifying a set of necessary conditions for the existence of deadlocks. This is the theory used by `Chord`, one of the most effective deadlock detection tools for `Java` according to [5, 16] (in the following we will compare `Chord` with our verifier).

Non-static approaches

Non static techniques usually analyse programs that have finite models. The advantage of these techniques is that they perform a very precise analysis.

Model Checking. [13] presents a model checking approach composed of two steps. A first step generates a so called *trace program* that records the critical concurrent operations and discards non critical parts. In a second step this program is analysed using an off-the-shelf model checker. Like this technique, our approach tries to get rid of non critical parts of the program. In our case, we use lams, thus allowing to perform the analysis in a reduced but key part of the program, without constraining the program model to be finite.

Monitoring Analysis. Monitoring techniques detect potential deadlocks at runtime. Our analysis of circularities has some similarities with the one presented in [3], such as the management of re-entrance. On the other hand the two approaches differ substantially when parallel codes is detected. Our technique detects the parallel states in the type system and, therefore, is over-approximate. The analysis of [3], being at runtime, tags each segment of the program that is reached by the execution flow by specify the exact order of lock acquisitions. Thereafter, they use an hybrid strategy for detecting potential deadlocks that might occur because of different scheduler choices (than the current one). The theory of [3] has been prototyped in the `GoodLock` tool, which has been used in the following assessment of our verifier. This technique has also been refined in `Sherlock`, which also uses symbolic executions [5].

JaDA

The technique described in this paper has been prototyped. The verifier, called `JaDA` [8, 9], is implemented in `Java` and includes features such as constructors, arrays, exceptions, static members, interfaces, inheritance, recursive data types (Garcia's PhD thesis contains the technical details of these extensions [7]). These extensions have made possible to deliver an initial assessment of `JaDA` with respect to existing deadlock analysis tools for `Java`. In particular, we have taken `Chord` for static analysis [16], `Sherlock` for dynamic analysis [5], and `GoodLock` for hybrid analysis [3]. We have also considered a commercial tool, `ThreadSafe` [4] [2]. Out of these tools, we were able to install and effectively test only two of them: `Chord` and `ThreadSafe`; the results corresponding to `GoodLock` and `Sherlock` come from [5]. We also had problems in testing `Chord` with some of the examples in the benchmarks, perhaps due to some misconfigurations, that we were not able to solve because `Chord` has been discontinued.

We have analysed a number of programs that exhibit a variety of sharing patterns. The source of all benchmarks in Table 1 is available either at [5, 16] or in the `JaDA-deadlocks` repository[5]. Since the current release of `JaDA` does not completely cover `JVM`, in order to gain preliminary experience, we modified the Java libraries and the multithreaded server programs of RayTracer, MolDyn and MonteCarlo (labelled with "(*)" in the Table 1) and implemented them in our system. This required little programming overhead; in particular, we removed volatile variables, avoided the use of `Runnable` interfaces for creating threads, and reduced the invocations of native methods involved in I/O operations.

[4] http://www.contemplateltd.com/threadsafe
[5] https://github.com/abelunibo/Java-Deadlocks

Table 1 Comparison with different deadlock detection tools. The inner cells show the number of deadlocks detected by each tool. The output labelled "(*)" are related to modified versions of the original programs: see the text.

	Static		Hybrid	Dynamic	Commercial
benchmarks	JaDA	Chord	GoodLock	Sherlock	ThreadSafe
Sor	1	1	7	1	4
RayTracer (*)	0	0	8	2	0
MolDyn (*)	0	0	6	1	0
MonteCarlo (*)	0	0	23	2	0
BuildNetwork	3	0			0
PhilosophersN	3	0			0
ThreadArrays	1	1			1
ThreadArraysWJoins	1	1			0
ScalaSimpleDeadlock	1				
ScalaPhilosophersN	3				

The first block of programs belongs to a well known group used as benchmarks for several `Java` analysis tools. In its current state `JaDA` only detects 1 deadlock in all of the four analysed programs from this group. It gives responses that are similar to `ThreadSafe` and `Chord` (`ThreadSafe` appears a bit more imprecise on Sor). The programs in the second block corresponds to examples designed to test our tool against complex deadlock scenarios like the `Network` program. We notice that both `Chord` and `ThreadSafe` fail to detect those kinds of deadlocks. The third group reports the analysis of two examples of `Scala` programs [17] (the `Scala` compiler 2.11 produces `Java` bytecode). We finally remark that, to the best of our knowledge, there is no static deadlock analysis tool for `Scala` (for this reason the entries corresponding to the other tools are empty).

8 Conclusions

We have defined a new technique for detecting deadlocks in `Java` programs by analysing the `Java` intermediate language `JVML`. The technique has been specified by focusing on a subset of `JVML` featuring thread creations and synchronizations, called `JVML`$_d$. We have also developed a prototype, called `JaDA`, which also covers complex features of `Java`, such as static members, arrays, recursive data types, exception handling, inheritance and dynamic dispatch. These extensions have made possible to deliver an initial assessment of `JaDA` with respect to existing deadlock analysis tools for `Java`.

Our future work includes the analysis of features of `Java` that have not yet been studied. One relevant feature is thread coordination, which is expressed by the methods `wait`, `notify` and `notifyAll`. The solution we are currently investigating uses more expressive *lams*, with special dependencies $(a, a^w)_t$ and $(a, a^n)_t$ for representing the wait-notify relations. In particular, $(a, a^w)_t$ means that "thread t has the lock a and has invoked the method `wait` on it"; $(a, a^n)_t$ means that "thread t has the lock a and has invoked the method `notify` on it". These dependencies *and* the analysis that the wait operation *happens-before* the matching notification should increase the precision in detecting deadlocks.

Another extension addresses *native methods*, namely methods that are not implemented within the language and that are used when it is necessary to interact with the Operating System or for meta-programming purposes. Our current solution is to manually insert in the BCT the behavioural types of native methods. We are investigating testing mechanisms that may help in writing the types of such methods.

References

1 Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *13th Australian Software Engineering Conference (ASWEC 2001)*, pages 68–75, 2001.

2 Robert Atkey and Donald Sannella. Threadsafe: Static analysis for Java concurrency. *ECEASST*, 72, 2015. URL: `http://journal.ub.tu-berlin.de/eceasst/article/view/1025`.

3 Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *in Hardware and Software Verification and Testing*, volume 3875 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2005.

4 Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of OOPSLA 2002*, pages 211–230. ACM, 2002.

5 Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE-22)*, pages 353–365. ACM, 2014.

6 Stephen N. Freund and John C. Mitchell. A type system for the Java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4):271–321, 2003.

7 Abel Garcia. *Static analysis of concurrent programs based on behavioral type systems*. PhD thesis, School in Computer Science and Engineering, 2017. Available at `JaDA.cs.unibo.it`.

8 Abel Garcia and Cosimo Laneve. The verifier `JaDA`. Available at `JaDA.cs.unibo.it`, 2016.

9 Abel Garcia and Cosimo Laneve. `JaDA` – the Java Deadlock Analyser. To appear as a chapter of the book “Behavioural Types for Reliable Large-Scale Software Systems”, available at `JaDA.cs.unibo.it`, 2017.

10 Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of 25th International Conference on Concurrency Theory CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.

11 Elena Giachino and Cosimo Laneve. Deadlock detection in linear recursive programs. In *14th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2014)*, volume 8483 of *Lecture Notes in Computer Science*, pages 26–64. Springer, 2014.

12 Futoshi Iwama and Naoki Kobayashi. A new type system for jvm lock primitives. In *Proc. of the ASIAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (ASIA-PEPM '02)*, pages 71–82, New York, NY, USA, 2002. ACM.

13 Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering*, pages 327–336. ACM, 2010.

14 Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017.

15 Cosimo Laneve. A type system for JVM threads. *Theoretical Computer Science*, 290(1):741 – 778, 2003.

16 Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE 2009)*, pages 386–396. ACM, 2009.

17 Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

18 Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the Java Virtual Machine Language: 1. In

Proceedings of the 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07), page 10. ACM, 2007.

19 Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

20 Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *Lecture Notes in Computer Science*, pages 602–629. Springer, 2005.

A The full JVML$_d$

The language we address in the Appendix is actually an extension of the one presented in Section 3 because

1. we also consider the instruction `join C` that pops the thread on top of the stack and joins it with the current one. This operation corresponds to `invokevirtual java/lang/Thread/join()` on a thread of class `C` in `JVML`;
2. we admit that methods update fields, not just constructors;
3. we admit `synchronized` methods.

These extensions (in particular the first two) will impact on the type system in a significative way because we need (*i*) to keep track of termination of threads and (*ii*) to analyze effects. In particular, as regards (*ii*), we must check data races of parallel threads (for example, one thread reading an object and the other one writing on the object) because such races reduce the precision of the analysis.

B JVML$_d$ and its operational semantics

The instructions of JVML$_d$ perform a transformation of machine states, called *configurations* and noted

$$\Vdash_H \langle \varphi_1, z_1\rangle_{t_1} \cdots \langle \varphi_n, z_n\rangle_{t_n}$$

where H is the *heap* that contains objects created by `new` (see below) and a tuple $\langle \varphi, z\rangle_t$ represents an active thread t with:

- φ is a stack of activation records (or *frames*) (pc, f, s) with
  - pc is the *program counter* which contains the address of the instruction to be executed,
  - f is a total map from the set of local variables to the set of values,
  - s is the *operand Stack* of values;
- z is the set of objects locked by the thread.

The object in H contains:

- fields: the value of field `f` of object o is accessed with $H(o).\mathtt{f}$; the update is noted $H(o).\mathtt{f} \mapsto v$;
- the class of the object, returned by $H(o).class$;
- a counter $\mathtt{z}$ that records the number of locks performed by a thread on the object.

We use the notation $H(o) \mapsto (\rho_\perp^{\mathtt{C}}, \mathtt{C})$ to allocate heap space for the object o of class `C`, by assigning a default value $\rho_\perp^{\mathtt{C}}$ to the fields and setting $H(o).\mathtt{z} = 0$. The operational semantics of JVML$_d$ is presented in Figures 5 and 6.

The *initial configuration* of a JVML$_d$ program P with main method $\mathtt{C}.main$ is $\Vdash_H \langle (1^{\mathtt{C}.main}, f_\perp[0 \mapsto main], \epsilon), \epsilon\rangle_{main}$, where H defines the object $main$ and $f_\perp$ is the function that is always undefined.

In the following we will use $\mathscr{C}$ to range over sets

$$\langle (pc^{\mathtt{C.m}}, f_1, s_1) \cdot \varphi_1, z_1\rangle_{t_1} \cdots \langle (pc^{\mathtt{D.m}}, f_n, s_n) \cdot \varphi_n, z_n\rangle_{t_n}$$

and configurations will be ranged over by $\Vdash_H \mathscr{C}$.

(S-POP)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{pop}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, v \cdot s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z\rangle_t}$$

(S-PUSH)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{push}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, 0 \cdot s) \cdot \varphi, z\rangle_t}$$

(S-INC)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{inc}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, n \cdot s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, (n+1) \cdot s) \cdot \varphi, z\rangle_t}$$

(S-LOAD)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{load}\ x}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, f(x) \cdot s) \cdot \varphi, z\rangle_t}$$

(S-STORE)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{store}\ x}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, v \cdot s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f[x \mapsto v], s) \cdot \varphi, z\rangle_t}$$

(S-GOTO)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{goto}\ L^{\mathtt{C.m}}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (L^{\mathtt{C.m}}, f, s) \cdot \varphi, z\rangle_t}$$

(S-PUTFIELD)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{putfield}\ \mathtt{D}.a\ \tau \qquad H' = H(o).a \mapsto e}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, e \cdot o \cdot s) \cdot \varphi, z\rangle_t \to \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z\rangle_t}$$

(S-GETFIELD)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{getfield}\ \mathtt{D}.a\ \tau \qquad H(o).a = e}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, e \cdot s) \cdot \varphi, z\rangle_t}$$

(S-NEW)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{new}\ \mathtt{D} \qquad o \notin dom(H) \qquad H' = H[o \mapsto (\rho^D, D)]}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z\rangle_t \to \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, o \cdot s) \cdot \varphi, z\rangle_t}$$

(S-IF-TRUE)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{if}\ L^{\mathtt{C.m}} \qquad n \neq 0}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, n \cdot s) \cdot \varphi, z\rangle_t \to \Vdash_H \langle (L^{\mathtt{C.m}}, f, s) \cdot \varphi, z\rangle_t}$$

(S-IF-FALSE)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{if}\ L^{\mathtt{C.m}}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, 0 \cdot s) \cdot \varphi, z)\rangle_t \to \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z\rangle_t}$$

Figure 5 Reduction rules: Basic operators

C Correctness

Let $\mathscr{A}$ be the (infinite) set of *object names*, ranged over by a, b, c, $\cdots$. In the syntax of ℓ, see Section 4, the operations "&" and "+" are associative, commutative with $\mathtt{0}$ being the identity on &, and definitions and lams are equal up-to alpha renaming of bound names. Namely, if $a \notin var(\ell)$, the following axioms hold:

$$(\nu\, a)\ell = \ell \qquad ((\nu\, a)\ell')\&\ell = (\nu\, a)(\ell'\&\ell) \qquad ((\nu\, a)\ell') + \ell = (\nu\, a)(\ell' + \ell)$$

Additionally, when $\mathtt{V}$ ranges over lams that do not contain function invocations, the following axioms hold:

$$\mathtt{V}\&\mathtt{V} = \mathtt{V} \qquad \mathtt{V} + \mathtt{V} = \mathtt{V} \qquad \mathtt{V}\&(\ell' + \ell'') = \mathtt{V}\&\ell' + \mathtt{V}\&\ell'' \tag{1}$$

These axioms permit to rewrite a lam *without function invocations* as a *collection* (operation +) *of relations* (elements of a relation are gathered by the operation &). Let $\equiv$ be the least congruence containing the above axioms. (The axioms (1) are restricted to terms $\mathtt{V}$ that do not contain function invocations because $\mathtt{m}(\overline{\rho}) \to \rho'\&((a,b)_t + (b,c)_t) \neq (\mathtt{m}(\overline{\rho}) \to \rho'\&(a,b)_t) + (\mathtt{m}(\overline{\rho}) \to \rho'\&(b,c)_t)$ because the evaluation of the two lams (see below) may produce terms with different names.)

Operational semantics.

Let a *lam context*, noted $\mathtt{L}[\ ]$, be a term derived by the following syntax:

$$\mathtt{L}[\ ] \quad ::= \quad [\ ] \quad | \quad \ell\&\mathtt{L}[\ ] \quad | \quad \ell + \mathtt{L}[\ ]$$

As usual $\mathtt{L}[\ell]$ is the lam where the hole of $\mathtt{L}[\ ]$ is replaced by ℓ. According to the syntax, lam contexts have no ν-binder; that is, the hassle of name captures is avoided. The operational

(S-INVK)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{invokevirtual}\ \mathtt{D.m}'(\mathtt{T}_1, \dots, \mathtt{T}_n) \\ \mathtt{synchronized} \notin \mathit{mod}(\mathtt{D.m}') \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v_n \cdot \dots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \rangle_t \to \\ \Vdash_H \langle (1^{\mathtt{D.m}'}, f_2[0 \mapsto o, 1 \mapsto v_1, \dots, n \mapsto v_n], \epsilon) \cdot \\ (pc^{\mathtt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \rangle_t \end{array}}$$

(S-INVK-SYNCH-0)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{invokevirtual}\ \mathtt{D.m}'(\mathtt{T}_1, \dots, \mathtt{T}_n) \\ \mathtt{synchronized} \in \mathit{mod}(\mathtt{D.m}') \\ H(o).\mathbb{z} = 0 \quad H^i = H(o).\mathbb{z} \mapsto 1 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v_n \cdot \dots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \backslash \{o\} \rangle_t \to \\ \Vdash_H \langle (1^{\mathtt{D.m}'}, f_2[0 \mapsto o, 1 \mapsto v_1, \dots, n \mapsto v_n], \epsilon) \cdot \\ (pc^{\mathtt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \cup \{o\} \rangle_t \end{array}}$$

(S-INVK-SYNCH-N)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{invokevirtual}\ \mathtt{D.m}'(\mathtt{T}_1, \dots, \mathtt{T}_n) \\ \mathtt{synchronized} \in \mathit{mod}(\mathtt{D.m}') \\ H(o).\mathbb{z} = n,\ n > 0 \quad H' = H(o).\mathbb{z} \mapsto n + 1 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v_n \cdot \dots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \cup \{o\} \rangle_t \to \\ \Vdash_{H'} \langle (1^{\mathtt{D.m}'}, f_2[0 \mapsto o, 1 \mapsto v_1, \dots, n \mapsto v_n], \epsilon) \cdot \\ (pc^{\mathtt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \cup \{o\} \rangle_t \end{array}}$$

(S-START)
$$\dfrac{P[pc^{\mathtt{C.m}}] = \mathtt{start}\ \mathtt{D}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi, z \rangle_t \to \\ \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t\ , \\ \langle (1^{\mathtt{D}.run}, f_2[0 \mapsto o], \epsilon), \epsilon \rangle_o \end{array}}$$

(S-JOIN)
$$\dfrac{P[pc^{\mathtt{C.m}}] = \mathtt{join} \qquad P[pc^{\mathtt{D}.run}] = \mathtt{return}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi_{t_1}, z_{t_1} \rangle_t\ , \langle (pc^{\mathtt{D}.run}, f_2, s_2), z_{t_2} \rangle_o \to \\ \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi_{t_1}, z_{t_1} \rangle_t\ , \langle (pc^{\mathtt{D}.run}, f_2, s_2), z_{t_2} \rangle_o \end{array}}$$

(S-RETURN)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{return} \\ \mathtt{synchronized} \notin \mathit{mod}(\mathtt{C.m}) \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t \to \\ \Vdash_H \langle pc^{\mathtt{D.m}}, f_2, v \cdot s_2) \cdot \varphi, z \rangle_t \end{array}}$$

(S-RETURN-SYNCH-0)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{return} \qquad \mathtt{synchronized} \in \mathit{mod}(\mathtt{C.m}) \\ H(o).\mathbb{z} = 1 \quad H' = H(o).\mathbb{z} \mapsto 0 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \uplus \{o\} \rangle_t \to \\ \Vdash_{H'} \langle (pc^{\mathtt{D.m}}, v \cdot f_2, s_2) \cdot \varphi, z \rangle_t \end{array}}$$

(S-RETURN-SYNCH-N)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{return} \qquad \mathtt{synchronized} \in \mathit{mod}(\mathtt{C.m}) \\ H(o).\mathbb{z} = n, n > 1 \quad H' = H(o).\mathbb{z} \mapsto n - 1 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, \langle f_2, \bullet \cdot s_2) \cdot \varphi, z \cup \{o\} \rangle_t \to \\ \Vdash_{H'} \langle (pc^{\mathtt{D.m}}, f_2, v \cdot s_2 \cdot \varphi, z \cup \{o\} \rangle_t \end{array}}$$

(S-MONITOREXIT-0)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{monitorexit} \\ H(o).\mathbb{z} = 1 \quad H' = H(o).\mathbb{z} \mapsto 0 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \uplus \{o\} \rangle_t \to \\ \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t \end{array}}$$

(S-MONITOREXIT-N)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{monitorexit} \\ H(o).\mathbb{z} = n,\ n > 1 \\ H' = H(o).\mathbb{z} \mapsto n - 1 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \cup \{o\} \rangle_t \to \\ \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \cup \{o\} \rangle_t \end{array}}$$

(S-MONITORENTER-0)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{monitorenter} \\ H = H(o).\mathbb{z} = 0 \\ H' = H(o).\mathbb{z} \to 1 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \backslash \{o\} \rangle_t \to \\ \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \uplus \{o\} \rangle_t \end{array}}$$

(S-MONITORENTER-N)
$$\dfrac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{monitorenter} \\ H = H(o).\mathbb{z} = n,\ n > 0 \\ H' = H(o).\mathbb{z} \to n + 1 \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \cup \{o\} \rangle_t \to \\ \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \cup \{o\} \rangle_t \end{array}}$$

Figure 6 Reduction rules: Invocations and synchronizations

semantics of a program $\big(\mathscr{L}, \ell\big)$ is a transition system where *states* are lams, the *transition relation* is the least one satisfying the rule

$$\frac{\begin{array}{c}\mathtt{m}(\overline{\rho}) \to \rho' = (\nu\,\overline{c})\ell_{\mathtt{m}} \in \mathscr{L} \quad (\overline{\rho} \to \rho')\sigma = \overline{\rho}'' \to \rho''' \quad \overline{c}' \text{ fresh} \\ (\ell_{\mathtt{m}}\{^{\overline{c}'}/_{\overline{c}}\})\sigma = \ell'_{\mathtt{m}}\end{array}}{\mathtt{L}[\mathtt{m}(\overline{\rho}'') \to \overline{\rho}'''] \to \mathtt{L}[\ell'_{\mathtt{m}}]}$$

and the initial state is the lam ℓ' such that $\ell \equiv (\nu\,\overline{c})\ell'$ and ℓ' does not contain any ν-binder, as well as the lam $\ell_{\mathtt{m}}$. (The class name in the names of lam functions has been dropped, for simplicity.) We write $\to^*$ for the reflexive and transitive closure of $\to$.

By (RED), a lam ℓ is evaluated by successively replacing function invocations with the corresponding lam instances. Name creation is handled by replacing bound names of function bodies with fresh names.

Flattening and circularities

Informally, a lam has a *circularity* when it has a conjunction of dependencies whose transitive closure contains a pair $(a, a)_{\checkmark}$ (see Section 5). Here the hassle is that lams cannot be always rewritten in a form suitable for defining circularities, namely disjunctions of conjunctions, because the equations (1) do not apply to terms containing invocations. To overcome this problem, we define the operation of flattening. The transitive closure of a lam has been defined in Section 5.

▸ **Definition 1.** The *flattening* of a lam ℓ, noted $(\ell)^\flat$, is the lam ℓ where every function invocation has been replaced by $\mathtt{0}$.

For example, let $\ell = \mathtt{m}(a, b, c, t) + (a, b)_t \& \mathtt{m}'(b, c, t') \& \mathtt{m}(d, b, c, t)$ (we assume that return types of lam functions are empty). Then $(\ell)^\flat = \mathtt{0} + (a, b)_t \& \mathtt{0} \& \mathtt{0}$.

▸ **Definition 2.** A lam ℓ has a *circularity* if $(\ell)^\flat \equiv \sum_{i \in I} \ell_i$, where every ℓ_i is a conjunction of dependencies, and, for some a and i, $(a, a)_{\checkmark}$ is a subterm of ${\ell_i}^+$.

A lam program $\big(\mathscr{L}, \ell\big)$ *has a circularity* if there exists ℓ' such that $\ell \to {}^*\ell'$ and ℓ' has a circularity.

Later stage relation

In the following subject reduction theorem we use a relation, called *later-stage relation* $\succcurlyeq$, which is the least congruence with respect to lams that contains the rules presented in Figure 7. The notation $\succcurlyeq$ assumes the presence of a behavioural class table BCT – we should have noted the later-stage relation as $\succcurlyeq_{\text{BCT}}$, but we prefer to keep the BCT implicit. A relevant property of the later stage relation is the following.

▸ **Lemma 3** (Circularities). *Let $\ell \succcurlyeq \ell'$. If ℓ' has a circularity then ℓ has also a circularity.*

This result mostly follows from [10, 14].

C.1 Static-time typing

We rewrite the rules of static time typing. They are more verbose than those of Section 4 because we cover more features than there. However, the reader should easily retrieve the rules in the main paper by overlooking at the new items.

Since methods may update fields then parallel threads may manifest data races that jeopardise the precision of the analysis. This means that, in order to deliver a more precise

$$\text{(L-0)}\quad \ell \succcurlyeq 0 \qquad \text{(L-RES)}\ \frac{\ell \succcurlyeq \ell'}{(\nu a)\ell \succcurlyeq (\nu a)\ell'} \qquad \text{(L-PLUS)}\ \frac{\ell_1 \succcurlyeq \ell_1' \quad \ell_2 \succcurlyeq \ell_2'}{\ell_1 + \ell_1' \succcurlyeq \ell_2 + \ell_2'} \qquad \text{(L-AND)}\ \frac{\ell_1 \succcurlyeq \ell_1' \quad \ell_2 \succcurlyeq \ell_2'}{\ell_1 \& \ell_1' \succcurlyeq \ell_2 \& \ell_2'}$$

$$\text{(L-INVK)}\ \frac{\text{BCT}(\texttt{C.m})(\bar\rho, t, a) = (\nu\, \overline{a'})\langle \rho, \mathscr{T}, \mathscr{R}, K, \overline{\rho'}, \ell\rangle \quad \bar b \textit{ fresh}}{\texttt{C.m}(\bar\rho, t, a) \to (\rho\{^{\bar b}/_{\overline{a'}}\}) \succcurlyeq \ell\{^{\bar b}/_{\overline{a'}}\}}$$

Figure 7 The later-stage relation

analysis, we need to analyze effects of methods and verify their consistency when they run in parallel. For this reason, we annotate fields with *accesses* $\texttt{h} = \{\texttt{-}, \texttt{r}, \texttt{w}\}$ – with _ meaning no access, $\texttt{r}$ meaning read access, $\texttt{w}$ meaning write (and possible read) access – and we let $\texttt{-} \leqslant \texttt{r} \leqslant \texttt{w}$. Since $(\texttt{h}, \leqslant)$ is a lattice, we will use the operation of least upper bound $\sqcup$. In the following, the *flattened record types* ς are terms where fields are labelled with accesses:

$$\tau \ ::= \ \top \mid \texttt{int} \mid X \mid a \qquad\qquad \varsigma \ ::= \ ([\overline{\texttt{f}^{\texttt{h}} : \tau}], \texttt{C})$$

With an abuse of notation, we will range over flattened record types with $\tau, \tau', \cdots$. We always shorten $\texttt{f}^{\texttt{-}}$ into $\texttt{f}$.

Let

$$\texttt{f}^{\texttt{h}} : \tau + \texttt{f}^{\texttt{h}'} : \tau' \ \stackrel{def}{=} \ \begin{cases} \texttt{f}^{\texttt{h}\sqcup\texttt{h}'} : \tau & \text{if } \texttt{h}, \texttt{h}' \leqslant \texttt{r} \text{ and } \tau = \tau' \\ \texttt{f}^{\texttt{w}} : \texttt{int} & \text{if } \texttt{h} \sqcup \texttt{h}' = \texttt{w} \text{ and } \tau = \texttt{int} = \tau' \\ \texttt{f}^{\texttt{w}} : \top & \text{otherwise} \end{cases}$$

It is worth to notice that $\texttt{f}^{\texttt{h}} : X + \texttt{f}^{\texttt{h}'} : Y$ is $\top$ when $X \neq Y$ (and similarly when $X = a$, $Y = b$ and $a \neq b$). Let also

$$([\cdots, \texttt{f}_i^{\texttt{h}_i} : \tau_i, \cdots], \texttt{C}) + ([\cdots, \texttt{f}_i^{\texttt{h}_i'} : \tau_i', \cdots], \texttt{C}) \ \stackrel{def}{=} \ ([\cdots, \texttt{f}_i^{\texttt{h}_i} : \tau_i + \texttt{f}_i^{\texttt{h}_i'} : \tau_i', \cdots], \texttt{C})$$

(it is assumed that the fields of the records are the same – because they have the same class type – and the operation is performed on every field).

Environments.

The foregoing static semantics uses abstract heaps, called *environments* Γ, which map names to type values or to flattened record types. There are two basic operations on environments: one for *sequential composition* – the *update* $\Gamma[\Gamma']$ – and one for *parallel composition* – the *merge* $\Gamma + \Gamma'$. They are defined as follows

$$\Gamma[\Gamma'](a) \ = \ \begin{cases} \Gamma(a) & \text{if } a \in \mathit{dom}(\Gamma)\backslash\mathit{dom}(\Gamma') \\ \Gamma'(a) & \text{if } a \in \mathit{dom}(\Gamma')\backslash\mathit{dom}(\Gamma) \\ ([\cdots, \texttt{f}_i^{\texttt{h}_i \sqcup \texttt{h}_i'} : \tau_i', \cdots], \texttt{C}) & \text{if } \Gamma(a) = ([\cdots, \texttt{f}_i^{\texttt{h}_i} : \tau_i, \cdots], \texttt{C}) \\ & \text{and } \Gamma'(a) = ([\cdots, \texttt{f}_i^{\texttt{h}_i'} : \tau_i', \cdots], \texttt{C}) \end{cases}$$

$$(\Gamma + \Gamma')(a) \ = \ \begin{cases} \Gamma(a) & \text{if } a \in \mathit{dom}(\Gamma)\backslash\mathit{dom}(\Gamma') \\ \Gamma'(a) & \text{if } a \in \mathit{dom}(\Gamma')\backslash\mathit{dom}(\Gamma) \\ \Gamma(a) + \Gamma'(a) & \text{otherwise} \end{cases}$$

For example, let $\Gamma = a \mapsto ([\mathtt{f}^{\mathtt{h}} : b], \mathtt{C})$ and $\Gamma' = a \mapsto ([\mathtt{f}^{\mathtt{w}} : c], \mathtt{C})$. Then $\Gamma[\Gamma'] = a \mapsto ([\mathtt{f}^{\mathtt{w}} : c], \mathtt{C})$, which is the standard update of an environment plus the recording of the writing operation, while $\Gamma + \Gamma' = a \mapsto ([\mathtt{f}^{\mathtt{w}} : \top], \mathtt{C})$, namely the field value is undefined because a *race condition* on the field $\mathtt{f}$ of a caused by the parallel execution of two threads. (This case was not possible in Section 4 because of the restriction that fields are read-only.)

In the following we will also use record types with fields labelled with accesses. We will range over them with $\rho, \rho', \cdots$ and the syntax is

$$\rho \quad ::= \quad \top \mid \mathtt{int} \mid X \mid (a[\overline{\mathtt{f}^{\mathtt{h}} : \rho}], \mathtt{C}) \ .$$

The operation $+$ is extended to record types as follows

$$\rho + \rho' \stackrel{def}{=} \begin{cases} (a[\cdots, \mathtt{f}^{\mathtt{h}} : \vartheta_{\mathtt{f}} + \mathtt{f}^{\mathtt{h}'} : \vartheta'_{\mathtt{f}}, \cdots], \mathtt{C}) & \text{if } \vartheta = (a[\cdots, \mathtt{f}^{\mathtt{h}} : \vartheta_{\mathtt{f}}, \cdots], \mathtt{C}) \\ & \text{and } \vartheta' = (a[\cdots, \mathtt{f}^{\mathtt{h}'} : \vartheta'_{\mathtt{f}}, \cdots], \mathtt{C}) \\ \top & \text{if } \mathit{root}(\vartheta) \neq \mathit{root}(\vartheta') \end{cases}$$

Finally, in the following we will use an extension of method record type that also contain the set of threads that have been joined by the method. Therefore, a *behavioural class table* BCT is a map from pairs $\mathtt{C.m}$ to *method types*, which are terms

$$(\bar{\rho}, t, a) \to (\nu\, \overline{a'})\langle \rho, \mathscr{T}, \mathscr{R}, K, \overline{\rho'}, \ell \rangle$$

where all the element are as in Section 4, except for K, which is new. This set contains thread names that occur in the arguments (in $\bar{\rho}$) and that have been synchronized within the body of $\mathtt{C.m}$. Correspondingly, we redefine $\bigsqcup$ as the commutative and associative operator defined by

$$\begin{array}{rcl} (\rho, Z_i, \mathscr{T}_i, \mathscr{R}_i, K_i, \Gamma_i) \bigsqcup \varnothing & \stackrel{def}{=} & (\rho, Z_i, \mathscr{T}_i, \mathscr{R}_i, K_i, \Gamma_i) \\ (\rho, Z_i, \mathscr{T}_i, \mathscr{R}_i, \Gamma_i) \bigsqcup (\rho, Z_i, \mathscr{T}_j, \mathscr{R}_j, K_j, \Gamma_j) & \stackrel{def}{=} & (\rho, Z_i, \mathscr{T}_i \cup \mathscr{T}_j, \mathscr{R}_i \cup \mathscr{R}_j, K_i \cap K_j, \Gamma_i + \Gamma_j) \end{array}$$

($\bigsqcup$ is only defined on tuples with the elements ρ and Z_i equal). We observe that the operation $\bigsqcup$ when applied to sets K returns their intersection.

C.2 Run-time typing

In the run-time typing rule we use environments that are *sequences of sequences of environments* used at static time. For example we may have $(\Gamma^{1,1} \cdots \Gamma^{1,n}) \cdot (\Gamma^{2,1} \cdots \Gamma^{2,m})$ (we remind that $\Gamma^{i,j}$ are vectors of maps, which are indexed by addresses. With an abuse of notation, we will range over run-time environments with $\Gamma, \Gamma', \cdots$. Whenever Γ is a run-time environment and we write $x \in \mathit{dom}(\Gamma)$ we mean that there is a map in Γ such that x belongs to the domain of the map. If we write $\Gamma(o) = ([\mathtt{f}_1^{\mathtt{h}_1} : \tau_1, \cdots, \mathtt{f}_i^{\mathtt{h}_i} : \tau_i, \cdots, \mathtt{f}_n^{\mathtt{h}_n} : \tau_n], \mathtt{C})$ we mean that $o \in \mathit{dom}(\Gamma)$ and every map γ in Γ such that $o \in \mathit{dom}(\gamma)$ satisfies $\gamma(o) = ([\mathtt{f}_1^{\mathtt{h}_1} : \tau_1, \cdots, \mathtt{f}_i^{\mathtt{h}_i} : \tau_i, \cdots, \mathtt{f}_n^{\mathtt{h}_n} : \tau_n], \mathtt{C})$.

▸ **Definition 4** (Γ-H Agreement)**.** An environment Γ *agrees with* a heap H, written $\Gamma \approx H$ if

- $o \in \mathit{dom}(H)$ implies $o \in \mathit{dom}(\Gamma)$ and
- $H(o).\mathit{class} = \mathtt{C}$ and $H(o).\mathtt{f}_i \mapsto v_i$, for $i \in 1..n$, imply $\Gamma(o) = ([\mathtt{f}_1^{\mathtt{h}_1} : \tau_1, \cdots, \mathtt{f}_i^{\mathtt{h}_i} : \tau_i, \cdots, \mathtt{f}_n^{\mathtt{h}_n} : \tau_n], \mathtt{C})$, where, if v_i is an object name, then $v_i = \tau_i$.

▸ **Definition 5** (S/F/Z-Agreements)**.** Given Γ, we define the following agreements between the static environments S, F and Z and the corresponding dynamic ones s, f, and z (Γ is always omitted in the notation).

Method invocations

$$\frac{\begin{array}{c} P[i] = \mathtt{invokevirtual}\ \ \mathtt{C.m}\ (\mathtt{T}_0, \cdots, \mathtt{T}_k) \qquad i+1 \in \mathit{dom}(P) \qquad S_i = \tau_k \cdots\ \tau_0 \cdot S' \\ \mathit{typeof}(\Gamma_i, \tau_0) = \mathtt{C} \quad \mathit{mk\_tree}(\Gamma_i, (\tau_0, \cdots, \tau_k)) = (\rho_0, \cdots, \rho_k) \\ \overline{b} = \mathit{names}(i) \quad \overline{b} \cap \mathit{var}(\rho_0, \cdots, \rho_k, t, \lceil Z_i \rceil) = \varnothing \\ \textsc{bct}(\mathtt{C.m})(\rho_0, \cdots, \rho_k, t, \lceil Z_i \rceil)(\overline{b}) = \langle \rho, \mathscr{T}', \mathscr{R}', K', (\rho'_0, \cdots, \rho'_k), \ell \rangle \\ \Gamma_{i+1} = \Gamma_i[\mathit{env}(\vartheta) + (+_{i \in 0..k}\ \mathit{env}(\rho'_i))] \\ S_{i+1} = \mathit{root}(\rho) \cdot S' \quad F_{i+1} = F_i \quad Z_{i+1} = Z_i \quad K_{i+1} = K_i \cup K' \\ \mathscr{T}_{i+1}, \mathscr{R}_{i+1} = \left\{ \begin{array}{ll} (\mathscr{T}_i \backslash K') \cup \mathscr{T}',\ (\mathscr{R}_i \backslash K') \cup \mathscr{R}' & \text{if } i \text{ is not recursive and } (\mathscr{T}_i \cup \mathscr{R}_i) \cap (\mathscr{T}' \cup \mathscr{R}') = \varnothing \\ \mathscr{T}_i \backslash K',\ (\mathscr{R}_i \backslash K') \cup \mathscr{R}' \cup \mathscr{T}' & \text{otherwise} \end{array} \right. \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : \mathtt{C.m}(\rho_0, \cdots, \rho_k, t, \lceil Z_i \rceil) \to \rho}$$

$$\frac{\begin{array}{c} P[i] = \mathtt{start}\ \mathtt{C} \qquad i+1 \in \mathit{dom}(P) \qquad S_i = t' \cdot S_{i+1} \\ \mathit{typeof}(\Gamma_i, t') = \mathtt{C} \quad \mathit{mk\_tree}(\Gamma_i, t') = \rho \\ \overline{b} = \mathit{names}(i) \quad \overline{b} \cap \mathit{var}(\rho, t', \mathit{lock}_{t'}) = \varnothing \\ \textsc{bct}(\mathtt{C.run})(\rho, t', \mathit{lock}_{t'})(\overline{b}) = \langle \mathit{void}, \mathscr{T}', \mathscr{R}', K', \rho', \ell \rangle \\ \Gamma_{i+1} = \Gamma_i + \mathit{env}(\rho') \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \quad K_{i+1} = K_i \cup K' \\ \mathscr{T}_{i+1}, \mathscr{R}_{i+1} = \left\{ \begin{array}{ll} \mathscr{T}_i \cup \{\tau\} \cup \mathscr{T}',\ \mathscr{R}_i \cup \mathscr{R}' & \text{if } i \text{ is not recursive} \\ & \text{and } \tau \notin \mathscr{T}_i \cup \mathscr{R}_i \\ \mathscr{T}_i,\ \mathscr{R}_i \cup \mathscr{R}' \cup \mathscr{T}' \cup \{\tau\} & \text{otherwise} \end{array} \right. \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : 0}$$

$$\frac{\begin{array}{c} P[i] = \mathtt{join}\ \mathtt{C} \qquad i+1 \in \mathit{dom}(P) \qquad S_i = t' \cdot S_{i+1} \\ \mathit{typeof}(\Gamma_i, t') = \mathtt{C} \quad \mathit{mk\_tree}(\Gamma_i, t') = \rho \\ \mathit{joined}(\mathtt{C.run}, \rho, t', \mathit{lock}_{t'}) = K' \quad \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \\ Z_i = Z_{i+1} \quad K_{i+1} = \left\{ \begin{array}{ll} K_i \cup K' & \text{if } \tau \in \mathscr{T}_i \cup \mathscr{R}_i \\ K_i \cup K' \cup \{\tau\} & \text{if } \tau \notin \mathscr{T}_i \cup \mathscr{R}_i \end{array} \right. \\ \mathscr{T}_{i+1} = (\mathscr{T}_i \backslash K') \backslash \{t'\} \quad \mathscr{R}_{i+1} = (\mathscr{R}_i \backslash K') \backslash \{t'\} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : (\lceil Z_i \rceil, \mathit{lock}_{t'})_t}$$

Locking instructions

$$\frac{\begin{array}{c} P[i] = \mathtt{monitorenter} \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \\ Z_{i+1} = a \cdot Z_i \quad K_i = K_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : 0}$$

$$\frac{\begin{array}{c} P[i] = \mathtt{monitorexit} \qquad i+1 \in \mathit{dom}(P) \\ \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \\ Z_{i+1} = Z_i \backslash a \quad K_i = K_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : 0}$$

Object manipulation instructions

$$\frac{\begin{array}{c} P[i] = \mathtt{new}\ \mathtt{C} \qquad i+1 \in \mathit{dom}(P) \quad a = \mathit{names}(i) \quad \mathsf{fields}(\mathtt{C}) = \bar{\mathtt{f}} \qquad \Gamma_{i+1} = \Gamma_i[a \mapsto ([\overline{\mathtt{f} : \top}], \mathtt{C})] \\ F_i = F_{i+1} \quad S_{i+1} = a \cdot S_i \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P :\ \widehat{\mathscr{T}_i}\ \&\ \check{\mathscr{R}_i}\ \&\ \widehat{Z_i}^{\,t}}$$

$$\frac{\begin{array}{c} P[i] = \mathtt{putfield}\ \mathtt{C.f} : \mathtt{T} \qquad i+1 \in \mathit{dom}(P) \\ S_i = \tau \cdot a \cdot S_{i+1} \quad \Gamma_i(a) = [\mathtt{f}^{\mathtt{h}} : \tau', \overline{\mathtt{f}^{\mathtt{h}} : \tau'}] \\ \Gamma_{i+1} = \Gamma_i[a \mapsto [\mathtt{f}^{\mathtt{w}} : \tau, \overline{\mathtt{f}^{\mathtt{h}} : \tau'}]] \quad F_i = F_{i+1} \\ K_i = K_{i+1} \quad Z_i = Z_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : 0}$$

$$\frac{\begin{array}{c} P[i] = \mathtt{getfield}\ \mathtt{C.f} : \mathtt{T} \qquad i+1 \in \mathit{dom}(P) \\ S_i = a \cdot S' \quad \Gamma_i(a) = [\mathtt{f}^{\mathtt{h}} : \tau, \overline{\mathtt{f}^{\mathtt{h}} : \tau}] \\ \Gamma_{i+1} = \Gamma_i[a \mapsto [\mathtt{f}^{\mathtt{h} \sqcup \mathtt{r}} : \tau, \overline{\mathtt{f}^{\mathtt{h}} : \tau}]] \\ S_{i+1} = \tau \cdot S' \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \\ K_i = K_{i+1} \quad \mathscr{T}_i = \mathscr{T}_{i+1} \quad \mathscr{R}_i = \mathscr{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : 0}$$

Method definitions

$$\frac{\begin{array}{c} \textsc{bct}(\mathtt{C.m}) = (\rho_0, \cdots, \rho_k, t, a) \to (\nu \overline{a}')\langle \rho, \mathscr{T}', \mathscr{R}', K', \overline{\rho'}, \ell \rangle \\ F_1 = F_\top[0 \mapsto \mathit{root}(\rho_0), \cdots, k \mapsto \mathit{root}(\rho_k)] \\ \Gamma_1 = +_{i \in 0..k}\ \mathit{env}(\rho_i) \\ S_1 = \varepsilon \quad Z_1 = a \quad \mathscr{T}_1 = \varnothing \quad \mathscr{R}_1 = \varnothing \quad K_1 = \varnothing \\ \Big(\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : \ell_i\ ;\ \Psi_i \Big)^{i \in \mathit{dom}(P)} \\ \ell = \sum_{i \in \mathit{dom}(P)} \ell_i\ \&\ \widehat{\mathscr{T}_i}\ \&\ \check{\mathscr{R}_i}\ \&\ \widehat{Z_i}^{\,t} \\ \bigsqcup_{i \in \mathit{dom}(P)} \Psi_i = (\rho, a, \mathscr{T}', \mathscr{R}', K', \Gamma') \\ \overline{\rho'} = \mathit{mk\_tree}(\Gamma', (\mathit{root}(\rho_0), \cdots, \mathit{root}(\rho_k))) \\ \overline{a'} = \mathit{fn}(\rho, \mathscr{T}', \mathscr{R}', K', \overline{\rho'}) \backslash\ \mathit{fn}(\rho_0, \cdots, \rho_k, t, a) \end{array}}{\textsc{bct} \vdash \mathtt{T}\ \mathtt{C.m}\ (\mathtt{T}_1, \cdots, \mathtt{T}_k)\ P}$$

$$\frac{\begin{array}{c} \textsc{bct}(\mathtt{C.m}) = (a'[\overline{\mathtt{f} : \rho}], \rho_1, \cdots, \rho_k, t, a) \to (\nu \overline{a''})\langle \rho, \mathscr{T}', \mathscr{R}', K', \overline{\rho'}, \ell \rangle \\ F_1 = F_\top[0 \mapsto a', 1 \mapsto \mathit{root}(\rho_1), \cdots, k \mapsto \mathit{root}(\rho_k)] \\ \Gamma_1 = \mathit{env}(a'[\overline{\mathtt{f} : \rho}]) \oplus (\textstyle\bigoplus_{j \in 1..k} \mathit{env}(\rho_j)) \\ S_1 = \varepsilon \quad Z_1 = a' \cdot a \quad \mathscr{T}_1 = \varnothing \quad \mathscr{R}_1 = \varnothing \quad K_1 = \varnothing \\ \Big(\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P : \ell_i\ ;\ \Psi_i \Big)^{i \in \mathit{dom}(P)} \\ \ell = \sum_{i \in \mathit{dom}(P)} \ell_i\ \&\ \widehat{\mathscr{T}_i}\ \&\ \check{\mathscr{R}_i}\ \&\ \widehat{Z_i}^{\,t} \\ \bigsqcup_{i \in \mathit{dom}(P)} \Psi_i = (\rho, a' \cdot a, \mathscr{T}', \mathscr{R}', K', \Gamma') \\ \overline{\rho'} = \mathit{mk\_tree}(\Gamma', (a', \mathit{root}(\rho_1), \cdots, \mathit{root}(\rho_k))) \\ \overline{a''} = \mathit{fn}(\rho, \mathscr{T}', \mathscr{R}', K', \overline{\rho'}) \backslash\ \mathit{fn}(a'[\overline{\mathtt{f} : \rho}], \rho_1, \cdots, \rho_k, t, a) \end{array}}{\textsc{bct} \vdash \mathtt{synchronized}\ \mathtt{T}\ \mathtt{C.m}\ (\mathtt{T}_0, \cdots, \mathtt{T}_k)\ P}$$

Figure 8 Type rules for $\mathtt{JVML}_d$ programs – extension of those in Figure 3

S-agreement.

- $\epsilon \approx \epsilon$,
- $\tau \cdot S \approx \bullet \cdot s$, if $S \approx s$,
- $\tau \cdot S \approx v \cdot s$, if $S \approx s$ and $\Gamma(v) = \chi$.

F-agreement.

- $\varnothing \approx \varnothing$,
- $F[x \mapsto \tau] \approx f[x \mapsto v]$, if $F \approx_\Gamma f$ and $\Gamma(v) = \tau$.

Z-agreement.

- $\epsilon \approx \epsilon$,
- $o \cdot Z \approx o \cdot z$ if $Z \approx z$

Let $\varepsilon \backslash a = \varepsilon$ and

$$(o \cdot z)\backslash a \;=\; \begin{cases} z & \text{if } o = a \\ o \cdot (z \backslash a) & \text{otherwise} \end{cases}$$

Let also $z \backslash (a \cdot z')$ be $(z \backslash a)\backslash z'$.

▸ **Definition 6** (Reachable instructions set)**.** Let *reachable*(P, i), called the set of addresses reachable from the instruction $i \in dom(P)$, be the least set satisfying the following equations:

$$\begin{array}{lcll} reachable(P,i) & = & \varnothing & \text{if } P[i] = \texttt{return} \\ reachable(P,i) & = & \{L\} \cup reachable(P,L) & \text{if } P[i] = \texttt{goto L} \\ reachable(P,i) & = & \{i+1, L\} \cup reachable(P,i+1) \cup reachable(P,L) & \text{if } P[i] = \texttt{if L} \\ reachable(P,i) & = & \{i+1\} \cup reachable(P,i+1) & \text{otherwise} \end{array}$$

In order to type configurations, we need to introduce some notation. We will use sequences of elements $\mathscr{T}$, $\mathscr{R}$ and K that we will address by using the same notation. The h-th element of the sequence $\mathscr{T}$ will be noted $\mathscr{T}^{(h)}$. Let $\mathscr{P}$ be a sequence of elements whose h-th, noted $\mathscr{P}^{(h)}$, is $\Big((F^{(1,h)}, S^{(1,h)}) \cdots (F^{(n_h,h)}, S^{(n_h,h)}),\ Z^{(h)}\Big)$.

The typing judgement for configurations is

$$P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \mathscr{C}) : \ell$$

that is defined by the following two rules:

$$\frac{\Big(P, \textsc{bct}, \Gamma^{(h)}, \mathscr{P}^{(h)}, \mathscr{T}^{(h)}, \mathscr{R}^{(h)}, K^{(h)} \vdash \langle \varphi_h, z_h \rangle_{t_h} : \ell_h\Big)^{h \in 1..n} \qquad \Gamma \approx H}{P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash \Big(\Vdash_H \langle \varphi_1, z_1 \rangle_{t_1} \cdots \langle \varphi_n, z_n \rangle_{t_n}\Big) : \&_{i \in 1..n} \ell_i} \;\text{(T-CONF-AND)}$$

$$\frac{\begin{array}{c} \Big(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i;\ \Psi_i \qquad \ell'_i = \ell_i \;\&\; \widehat{\mathscr{T}'_i} \;\&\; \breve{\mathscr{R}}'_i \;\&\; \widehat{Z'^{\,t}_i}\Big)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})} \\ Z'_{pc^{\texttt{C.m}}} = \overline{a'} \cdot a \qquad P, \textsc{bct}, \Gamma, \mathscr{P}', \mathscr{T}, \mathscr{R}, K \vdash \langle \varphi,\ z \backslash \overline{a'} \rangle_t : \ell \\ \mathscr{P}' = \Big((F^{(1)}, S^{(1)}) \cdots (F^{(n)}, S^{(n)}),\ Z\Big) \qquad \mathscr{P} = \Big((F', S') \cdot (F^{(1)}, S^{(1)}) \cdots (F^{(n)}, S^{(n)}),\ Z' \cdot Z\Big) \\ Z'_{pc^{\texttt{C.m}}} \cdot Z \approx z \quad S'_{pc^{\texttt{C.m}}} \approx s \quad F'_{pc^{\texttt{C.m}}} \approx f \end{array}}{P, \textsc{bct}, \Gamma' \cdot \Gamma, \mathscr{P}, \mathscr{T}' \cdot \mathscr{T}, \mathscr{R}' \cdot \mathscr{R}, K' \cdot K \vdash \langle (pc^{\texttt{C.m}}, f, s) \cdot \varphi, z \rangle_t : \sum_{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})} \ell'_i + \ell} \;\text{(T-CONF-SINGLE)}$$

C.3 Subject Reduction

▸ **Lemma 7.** *Let* `C.m` *be a method of a* $\texttt{JVML}_d$ *program. If*

$$\textsc{bct}(\texttt{C.m})(\overline{\rho}, t, a) = (\nu\, \overline{a'})\langle \rho, \mathscr{T}, \mathscr{R}, K, \overline{\rho'}, \ell \rangle\ .$$

then

$$\Big(\textsc{bct}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K, i \vdash_t P[\texttt{C.m}] : \ell_i\ ;\ \Psi_i \qquad \ell'_i = \ell_i\ \&\ \widehat{\mathscr{T}_i}\ \&\ \widecheck{\mathscr{R}}_i\ \&\ \widehat{Z_i}^{t}\Big)^{i \in reachable(P[\texttt{C.m}], 1^{\texttt{C.m}})},$$

where

$$\begin{aligned}
&F_1 = F_\top[0 \mapsto root(\rho_0), \cdots, k \mapsto root(\rho_k)],\\
&\Gamma_1 = +_{i \in 0..k}\, env(\rho_i),\\
&S_1 = \varepsilon,\\
&Z_1 = A, \quad (\textit{if } \texttt{C.m} \textit{ is synchronized } A = root(\rho_0) \cdot a, \textit{ otherwise } A = a)\\
&\mathscr{T}_1 = \varnothing,\\
&\mathscr{R}_1 = \varnothing,\\
&K_1 = \varnothing,\\
&\ell \succcurlyeq \textstyle\sum_{i \in reachable(P[\texttt{C.m}], 1^{\texttt{C.m}})} \ell'_i,\\
&\textstyle\bigsqcup_{i \in reachable(P[\texttt{C.m}], 1^{\texttt{C.m}})} \Psi_i = (\rho, A, \mathscr{T}', \mathscr{R}', K', \Gamma'), \quad \mathscr{T}' \subseteq \mathscr{T}, \quad \mathscr{R}' \subseteq \mathscr{R}, \quad K' \supseteq K\\
&\overline{\rho'} = mk\_tree(\Gamma', (root(\rho_0), \cdots, root(\rho_k))), \textit{ and}\\
&\overline{a'} \supseteq fn(\rho, \mathscr{T}', \mathscr{R}', K', \overline{\rho'}) \setminus fn(\rho_0, \cdots, \rho_k, t, a).
\end{aligned}$$

▸ **Theorem 8** (Subject Reduction)**.** *If* $P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \mathscr{C}) : \ell$ *and* $\Gamma \approx H$ *and* $\Vdash_H \mathscr{C} \rightarrow \Vdash_{H'} \mathscr{C}'$*, then there exists* ℓ' *and* Γ'*,* $\mathscr{P}'$*,* $\mathscr{T}'$*,*$\mathscr{R}'$*,*K' *such that* $\Gamma' \approx H'$ *and* $P, \textsc{bct}, \Gamma', \mathscr{P}', \mathscr{T}', \mathscr{R}', K' \vdash (\Vdash_{H'} \mathscr{C}') : \ell'$ *and* $\ell \succcurlyeq \ell'$ *.*

Proof. (*Sketch*) By case analysis on the last reduction rule used.

Case `pop`**.** We have $P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \langle(pc^{\texttt{C.m}}, f, v \cdot s) \cdot \varphi, z\rangle_t) : \ell$ and

$$\textsc{(s-pop)}\quad \frac{P[pc^{\texttt{C.m}}] = \texttt{pop}}{P \Vdash_H \langle(pc^{\texttt{C.m}}, f, v \cdot s) \cdot \varphi, z\rangle_t \rightarrow \Vdash_H \langle(pc^{\texttt{C.m}} + 1, f, s) \cdot \varphi, z\rangle_t}$$

By the configuration typing rules, we get

$$\begin{gathered}
\Big(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i \qquad \ell'_i = \ell_i\ \&\ \widehat{\mathscr{T}'_i}\ \&\ \widecheck{\mathscr{R}'_i}\ \&\ \widehat{Z'_i}^{t}\Big)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})}\\
P, \textsc{bct}, \Gamma'', \mathscr{P}'', \mathscr{T}'', \mathscr{R}'', K'' \vdash \langle\varphi, z\rangle_t : \ell_\varphi \qquad \mathscr{P}'' = \Big(\overline{(F'', S'')}, Z''\Big)\\
Z'_{pc^{\texttt{C.m}}} = \overline{a'} \cdot a \quad \overline{a'} \cdot Z'' \approx z \quad S'_{pc^{\texttt{C.m}}} \approx v \cdot s \quad F'_{pc^{\texttt{C.m}}} \approx f
\end{gathered}$$

From $(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})}$ and Definition 6 we derive that

$$\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', pc^{\texttt{C.m}} \vdash_t P[\texttt{C.m}] : \ell_{pc^{\texttt{C.m}}}$$

and

$$(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)}\ .$$

Therefore $\ell = \ell'_{pc^{\texttt{C.m}}} + (\sum_{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)} \ell'_i) + \ell_\varphi$. By the typing rule for `pop`: $S'_{pc^{\texttt{C.m}}} = \tau \cdot S'_{pc^{\texttt{C.m}}+1}$; therefore, since $S'_{pc^{\texttt{C.m}}} \approx v \cdot s$ then $S'_{pc^{\texttt{C.m}}+1} \approx s$, by Definition 5.

We can apply again the configuration typing rules and we obtain $\ell' = (\sum_{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)} \ell'_i) + \ell_\varphi$, thus $\ell \succcurlyeq \ell'$.

<u>Case `invokevirtual`.</u> We have $P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \langle (pc^{\mathtt{C.m}}, f, v_n \cdot \cdots \cdot v_1 \cdot o \cdot s) \cdot \varphi, z \rangle_t) : \ell$ and one of three operational rules may have been applied: ($\textsc{s-invk}$), ($\textsc{s-invk-synch-0}$), or ($\textsc{s-invk-synch-n}$). Let us consider the former:

$$\textsc{(s-invk)}\quad \frac{\begin{array}{c} P[pc^{\mathtt{C.m}}] = \mathtt{invokevirtual}\ \mathtt{D.m}'(\mathtt{T}_1, \ldots, \mathtt{T}_n) \\ \mathtt{synchronized} \notin \mathit{mod}(\mathtt{D.m}') \end{array}}{\begin{array}{c} P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v_n \cdot \cdots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \rangle_t \to \\ \Vdash_H \langle (1^{\mathtt{D.m}'}, f_2[0 \mapsto o, 1 \mapsto v_1, \ldots, n \mapsto v_n], \epsilon)\ \cdot (pc^{\mathtt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \rangle_t \end{array}}$$

By the configuration typing rules, we get

$$\begin{array}{c} \left(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i \qquad \ell'_i = \ell_i \ \&\ \widehat{\mathscr{T}'_i}\ \&\ \widecheck{\mathscr{R}'_i}\ \&\ \widehat{Z'_i}^{t}\right)^{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}})} \\ P, \textsc{bct}, \Gamma'', \mathscr{P}'', \mathscr{T}'', \mathscr{R}'', K'' \vdash \langle \varphi, z \rangle_t : \ell_\varphi \quad \mathscr{P}'' = \left(\overline{(F'', S'')}, Z''\right) \\ Z'_{pc^{\mathtt{C.m}}} = \overline{a'} \cdot a \quad \overline{a'} \cdot Z'' \approx z \quad S'_{pc^{\mathtt{C.m}}} \approx v_n \cdot \cdots \cdot v_1 \cdot o \cdot s \quad F'_{pc^{\mathtt{C.m}}} \approx f \end{array}$$

From $(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i)^{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}$ and Definition 6 we derive that

$$\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', pc^{\mathtt{C.m}} \vdash_t P[\mathtt{C.m}] : \ell_{pc^{\mathtt{C.m}}}$$

and

$$\left(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i\right)^{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)}.$$

Therefore

$$\ell = \ell'_{pc^{\mathtt{C.m}}} + \Big(\sum_{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)} \ell'_i\Big) + \ell_\varphi. \tag{2}$$

We assume that `C.m` is recursive (this case and the one when $pc^{\mathtt{C.m}}$ is inside an iteration are the interesting cases). By ($\textsc{t-invk}$) typing rule:

$$\ell_{pc^{\mathtt{C.m}}} = \mathtt{D.m}'(\rho_1, \cdots, \rho_n, t, \lceil Z'_{pc^{\mathtt{C.m}}} \rceil) \to \rho,$$

where

$$\begin{array}{c} S'_{pc^{\mathtt{C.m}}} = \tau_n \cdots\ \tau_1 \cdot S''' \quad \mathit{typeof}(\Gamma_{pc^{\mathtt{C.m}}}, \tau_1) = \mathtt{D} \\ \mathit{mk\_tree}(\Gamma_{pc^{\mathtt{C.m}}}, (\tau_1, \cdots, \tau_n)) = (\rho_1, \cdots, \rho_n) \qquad \overline{b} = \mathit{names}(pc^{\mathtt{C.m}}) \\ \textsc{bct}(\mathtt{D.m}')(\rho_1, \cdots, \rho_n, t, \lceil Z'_{pc^{\mathtt{C.m}}} \rceil)(\overline{b}) = \langle \rho, \mathscr{T}''', \mathscr{R}''', K''', (\rho'_1, \cdots, \rho'_n), \ell_{\mathtt{D.m}'} \rangle \\ \Gamma_{pc^{\mathtt{C.m}}+1} = \Gamma_{pc^{\mathtt{C.m}}}[\mathit{env}(\rho) + (+_{i \in 1..n}\, \mathit{env}(\rho'_i))] \\ S'_{pc^{\mathtt{C.m}}+1} = \mathit{root}(\rho) \cdot S''' \quad F'_{pc^{\mathtt{C.m}}+1} = F'_{pc^{\mathtt{C.m}}} \quad Z'_{pc^{\mathtt{C.m}}+1} = Z'_{pc^{\mathtt{C.m}}} \quad K'_{pc^{\mathtt{C.m}}+1} = K'_{pc^{\mathtt{C.m}}} \cup K''' \\ \mathscr{T}'_{pc^{\mathtt{C.m}}+1}, \mathscr{R}'_{pc^{\mathtt{C.m}}+1} = (\mathscr{T}'_{pc^{\mathtt{C.m}}} \backslash K''') \cup \mathscr{T}''',\ (\mathscr{R}'_{pc^{\mathtt{C.m}}} \backslash K''') \cup \mathscr{R}''' \end{array}$$

By ($\textsc{t-invk}$) typing rule, $S'_{pc^{\mathtt{C.m}}} = \tau_n \cdots\ \tau_1 \cdot S'''$ and $S'_{pc^{\mathtt{C.m}}+1} = \mathit{root}(\rho) \cdot S'''$ so since $S'_{pc^{\mathtt{C.m}}} \approx v_n \cdot \cdots \cdot v_1 \cdot o \cdot s$ then $S'_{pc^{\mathtt{C.m}}+1} \approx \bullet \cdot s$.

Therefore we obtain that

$$P, \textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K' \vdash \langle (pc^{\mathtt{C.m}} + 1, f, \bullet \cdot s) \cdot \varphi, z \rangle_t : \ell_{pc^{\mathtt{C.m}}+1} \tag{3}$$

where $\ell_{pc^{\mathtt{C.m}}+1} = (\sum_{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)} \ell'_i) + \ell_\varphi$

By Lemma 7,

$$\left(\ \textsc{bct}, \Gamma^{\mathtt{D.m}'}, F^{\mathtt{D.m}'}, S^{\mathtt{D.m}'}, Z^{\mathtt{D.m}'}, \mathscr{T}^{\mathtt{D.m}'}, \mathscr{R}^{\mathtt{D.m}'}, K^{\mathtt{D.m}'}, i \vdash_t P[\mathtt{D.m}'] : \ell_i^{\mathtt{D.m}'}\ ;\ \Psi_i\ \right)^{i \in \mathit{reachable}(P[\mathtt{D.m}'], 1^{\mathtt{D.m}'})},$$

where

$$
\begin{array}{l}
F_1^{\mathtt{D.m}'} = F_\top^{\mathtt{D.m}'}[0 \mapsto \mathit{root}(\rho_1), \cdots, n \mapsto \mathit{root}(\rho_n)],\\
\Gamma_1^{\mathtt{D.m}'} = \mathop{+}_{i \in 1..n} \mathit{env}(\rho_i),\\
S_1^{\mathtt{D.m}'} = \varepsilon,\\
Z_1^{\mathtt{D.m}'} = \lceil Z'_{pc^{\mathtt{C.m}}} \rceil,\\
\mathscr{T}_1^{\mathtt{D.m}'} = \varnothing,\\
\mathscr{R}_1^{\mathtt{D.m}'} = \varnothing,\\
K_1^{\mathtt{D.m}'} = \varnothing,\\
\ell_{\mathtt{D.m}'} \succcurlyeq \ell'', \quad \text{where } \ell'' = \sum_{i \in \mathit{reachable}(P[\mathtt{D.m}'], 1^{\mathtt{D.m}'})} \ell_i'^{\mathtt{D.m}'} \ \& \ \widehat{\mathscr{T}_i^{\mathtt{D.m}'}} \ \& \ \check{\mathscr{R}}_i^{\mathtt{D.m}'} \ \& \ \widehat{Z_i^{\mathtt{D.m}'}}^{t},\\
\bigsqcup_{i \in \mathit{reachable}(P[\mathtt{D.m}'], 1^{\mathtt{D.m}'})} \Psi_i = (\rho, \lceil Z'_{pc^{\mathtt{C.m}}} \rceil, \mathscr{T}'^{\mathtt{D.m}'}, \mathscr{R}'^{\mathtt{D.m}'}, K'^{\mathtt{D.m}'}, \Gamma'^{\mathtt{D.m}'}),\\
\overline{\rho'} = \mathit{mk\_tree}(\Gamma'^{\mathtt{D.m}'}, (\mathit{root}(\rho_1), \cdots, \mathit{root}(\rho_n))), \text{ and}\\
\overline{b} = \mathit{fn}(\rho, \mathscr{T}'^{\mathtt{D.m}'}, \mathscr{R}'^{\mathtt{D.m}'}, K'^{\mathtt{D.m}'}, \overline{\rho'}) \setminus \mathit{fn}(\rho_1, \cdots, \rho_n, t, a).
\end{array}
$$

Thus

$$
\begin{array}{l}
P, \textsc{bct}, \Gamma^{\mathtt{D.m}'} \cdot \Gamma', F^{\mathtt{D.m}'} \cdot F', S^{\mathtt{D.m}'} \cdot S', Z^{\mathtt{D.m}'} \cdot Z', \mathscr{T}^{\mathtt{D.m}'} \cdot \mathscr{T}', \mathscr{R}^{\mathtt{D.m}'} \cdot \mathscr{R}', K^{\mathtt{D.m}'} \cdot K' \vdash\\
\qquad (\langle (1^{\mathtt{D.m}'}, f'[0 \mapsto o, 1 \mapsto v_1, \ldots, n \mapsto v_n], \epsilon) \cdot \ (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t) : \ell'' + \ell_{pc^{\mathtt{C.m}}+1}
\end{array}
$$

and $\ell \succcurlyeq \ell'' + \ell_{pc^{\mathtt{C.m}}+1}$.

The last step is to prove that $\Gamma^{\mathtt{D.m}'} \cdot \Gamma \approx H$. This follows from Definition 4 and the fact that $\Gamma \approx H$.

<u>**Case start.**</u> We have $P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi, z \rangle_t) : \ell$ and then

(s-start)

$$
\frac{P[pc^{\mathtt{C.m}}] = \mathtt{start\ D}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi, z \rangle_t \rightarrow\Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t \ , \langle (1^{\mathtt{D.run}}, f_2[0 \mapsto o], \epsilon), \epsilon \rangle_o}
$$

By the configuration typing rules, we obtain

$$
\begin{array}{c}
\left(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i \qquad \ell_i' = \ell_i \ \& \ \widehat{\mathscr{T}_i'} \ \& \ \check{\mathscr{R}}_i' \ \& \ \widehat{Z_i'}^{t}\right)^{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}\\
P, \textsc{bct}, \Gamma'', \mathscr{P}'', \mathscr{T}'', \mathscr{R}'', K'' \vdash \langle \varphi, z \rangle_t : \ell_\varphi \quad \mathscr{P}'' = \left(\overline{(F'', S'')}, Z''\right)\\
Z'_{pc^{\mathtt{C.m}}} = \overline{a'} \cdot a \quad \overline{a'} \cdot Z'' \approx z \quad S'_{pc^{\mathtt{C.m}}} \approx o \cdot s \quad F'_{pc^{\mathtt{C.m}}} \approx f
\end{array}
$$

From $(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i)^{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}$ and Definition 6 we derive that

$$
\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', pc^{\mathtt{C.m}} \vdash_t P[\mathtt{C.m}] : \ell_{pc^{\mathtt{C.m}}}
$$

and

$$
\left(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i\right)^{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)}.
$$

Therefore

$$
\ell = \ell'_{pc^{\mathtt{C.m}}} + \Big(\sum_{i \in \mathit{reachable}(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)} \ell_i' \Big) + \ell_\varphi. \tag{4}
$$

By (t-start) typing rule:

$$
\begin{array}{c}
S'_{pc^{\mathtt{C.m}}} = o \cdot S'_{pc^{\mathtt{C.m}}+1}\\
\mathit{typeof}(\Gamma'_{pc^{\mathtt{C.m}}}, o) = \mathtt{D} \quad \mathit{mk\_tree}(\Gamma'_{pc^{\mathtt{C.m}}}, o) = \rho \quad \overline{b} = \mathit{names}(pc^{\mathtt{C.m}})\\
\textsc{bct}(\mathtt{D.run})(\rho, o, \mathit{lock}_o)(\overline{b}) = \langle \mathit{void}, \mathscr{T}''', \mathscr{R}''', K''', \rho', \ell \rangle\\
\Gamma'_{pc^{\mathtt{C.m}}+1} = \Gamma'_{pc^{\mathtt{C.m}}} + \mathit{env}(\rho') \quad F'_{pc^{\mathtt{C.m}}} = F'_{pc^{\mathtt{C.m}}+1} \quad Z'_{pc^{\mathtt{C.m}}} = Z'_{pc^{\mathtt{C.m}}+1} \quad K'_{pc^{\mathtt{C.m}}+1} = K'_{pc^{\mathtt{C.m}}}\\
\mathscr{T}_{pc^{\mathtt{C.m}}+1}, \mathscr{R}_{pc^{\mathtt{C.m}}+1} = \left\{ \begin{array}{ll} \mathscr{T}_{pc^{\mathtt{C.m}}} \cup \{o\} \cup \mathscr{T}''', \ \mathscr{R}_{pc^{\mathtt{C.m}}} \cup \mathscr{R}''' & \text{if } i \text{ is not recursive}\\ & \text{and } o \notin \mathscr{T}_{pc^{\mathtt{C.m}}} \cup \mathscr{R}_{pc^{\mathtt{C.m}}}\\ \mathscr{T}_{pc^{\mathtt{C.m}}}, \ \mathscr{R}_{pc^{\mathtt{C.m}}} \cup \mathscr{R}''' \cup \mathscr{T}''' \cup \{o\} & \text{otherwise} \end{array} \right.
\end{array}
$$

Let $S'_{pc^{\mathtt{C.m}}} \approx o \cdot s_1$; then $S'_{pc^{\mathtt{C.m}}+1} \approx s_1$. Therefore we obtain that

$$P, \textsc{bct}, \Gamma' \cdot \Gamma'', \mathscr{P}', \mathscr{T}' \cdot \mathscr{T}'', \mathscr{R}' \cdot \mathscr{R}'', K' \cdot K'' \vdash \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t : \ell_{pc^{\mathtt{C.m}}+1} \tag{5}$$

where $\mathscr{P}' = \Big((F', S') \cdot \overline{(F'', S'')}, \overline{a'} \cdot Z'' \Big)$ and $\ell_{pc^{\mathtt{C.m}}+1} = (\sum_{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)} \ell'_i) + \ell_\varphi$. By Lemma 7,

$$\Big(\textsc{bct}, \Gamma^{\mathtt{D.run}}, F^{\mathtt{D.run}}, S^{\mathtt{D.run}}, Z^{\mathtt{D.run}}, T^{\mathtt{D.run}}, K^{\mathtt{D.run}}, i \vdash_t P[\mathtt{D.run}] : \ell_i^{\mathtt{D.run}} \,;\, \Psi_i \Big)^{i \in reachable(P[\mathtt{D.run}], 1^{\mathtt{D.run}})},$$

where

$$\begin{aligned}
&F_1^{\mathtt{D.run}} = F_\top^{\mathtt{D.run}}[0 \mapsto root(\rho)],\\
&\Gamma_1^{\mathtt{D.run}} = env(\rho),\\
&S_1^{\mathtt{D.run}} = \varepsilon,\\
&Z_1^{\mathtt{D.run}} = \lceil Z'_{pc^{\mathtt{C.m}}} \rceil,\\
&\mathscr{T}_1^{\mathtt{D.run}} = \varnothing,\\
&\mathscr{R}_1^{\mathtt{D.run}} = \varnothing,\\
&K_1^{\mathtt{D.run}} = \varnothing,\\
&\ell^{\mathtt{D.run}} = \textstyle\sum_{i \in reachable(P[\mathtt{D.run}], 1^{\mathtt{D.run}})} (\ell_i^{\mathtt{D.run}} \;\&\; \widehat{\mathscr{T}_i^{\mathtt{D.run}}} \;\&\; \widecheck{\mathscr{R}_i^{\mathtt{D.run}}} \;\&\; \widehat{Z_i^{\mathtt{D.run}}}^{t}),\\
&\textstyle\bigsqcup_{i \in reachable(P[\mathtt{D.run}], 1^{\mathtt{D.run}})} \Psi_i = (void, \lceil Z'_{pc^{\mathtt{C.m}}} \rceil, \mathscr{T}'^{\mathtt{D.run}}, \mathscr{R}'^{\mathtt{D.run}}, K'^{\mathtt{D.run}}, \Gamma'^{\mathtt{D.run}}),\\
&\vartheta = mk\_tree(\Gamma'^{\mathtt{D.run}}, o),\\
&\overline{b} = fn(void, \mathscr{T}'^{\mathtt{D.run}}, \mathscr{R}'^{\mathtt{D.run}}, K'^{\mathtt{D.run}}, \vartheta) \setminus fn(\rho, o, lock_o).
\end{aligned}$$

Thus, let $\mathscr{P}''' = \mathscr{P}' \cdot ((F^{\mathtt{D.run}}, S^{\mathtt{D.run}}), Z_1^{\mathtt{D.run}})$, then

$$\begin{aligned}
&P, \textsc{bct}, (\Gamma' \cdot \Gamma'') \cdot \Gamma^{\mathtt{D.run}}, \mathscr{P}''', (\mathscr{T}' \cdot \mathscr{T}'') \cdot \mathscr{T}^{\mathtt{D.run}}, (\mathscr{R}' \cdot \mathscr{R}'') \cdot \mathscr{R}^{\mathtt{D.run}}, (K' \cdot K'') \cdot K^{\mathtt{D.run}} \vdash\\
&\quad (\Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t \;,\; \langle (1^{\mathtt{D.run}}, f_2[0 \mapsto o], \epsilon), \epsilon \rangle_o) : \ell^{\mathtt{D.run}} \& \ell_{pc^{\mathtt{C.m}}+1}.
\end{aligned}$$

and $\ell \succcurlyeq \ell^{\mathtt{D.run}} \& \ell_{pc^{\mathtt{C.m}}+1}$. The last step, namely $(\Gamma' \cdot \Gamma'') \cdot \Gamma^{\mathtt{D.run}} \approx H$, is similar to the previous case.

Case `return`. We have

$$P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t) : \ell$$

The following rules may have been applied: (s-return), (s-return-run), (s-return-synch-0), and (s-return-synch-n). Let us consider the first one (the other ones are similar).

$$\text{(s-return)}\quad \frac{P[pc^{\mathtt{C.m}}] = \mathtt{return} \quad \mathtt{synchronized} \notin mod(\mathtt{C.m})}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle pc^{\mathtt{D.m}}, f_2, v \cdot s_2) \cdot \varphi, z \rangle_t}$$

By the configuration typing rules, we have

$$\begin{gathered}
\Big(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i \qquad \ell'_i = \ell_i \;\&\; \widehat{\mathscr{T}'_i} \;\&\; \widecheck{\mathscr{R}'_i} \;\&\; \widehat{Z'_i}^{t} \Big)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}\\
P, \textsc{bct}, \Gamma'', \mathscr{P}'', \mathscr{T}'', \mathscr{R}'', K'' \vdash \langle (pc^{\mathtt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t : \ell_\varphi \quad \mathscr{P}'' = \Big(\overline{(F'', S'')}, Z'' \Big)\\
Z'_{pc^{\mathtt{C.m}}} = \overline{a'} \cdot a \quad \overline{a'} \cdot Z'' \approx z \quad S'_{pc^{\mathtt{C.m}}} \approx v \cdot s_1 \quad F'_{pc^{\mathtt{C.m}}} \approx f_1
\end{gathered}$$

Therefore

$$\ell = (\sum_{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})} \ell'_i) + (\sum_{i \in reachable(P[\mathtt{D.m}], pc^{\mathtt{D.m}})} \ell'_i) + \ell_\varphi \tag{6}$$

Moreover, $S''_{pc^{\mathtt{D.m}'}} \approx \bullet \cdot s_2$ and $S''_{pc^{\mathtt{D.m}'}}$ has been set while typing the invocation of `C.m` as $S''_{pc^{\mathtt{D.m}'}} = root(\theta) \cdot S'''$, where $root(\theta)$ is the value returned by the method invocation, then $S''_{pc^{\mathtt{D.m}'}} \approx v \cdot s_2$.

Therefore we obtain

$$\ell' = (\sum_{i \in reachable(P[\mathtt{D.m}], pc^{\mathtt{D.m}})} \ell'_i) + \ell_\varphi \tag{7}$$

then $\ell \succcurlyeq \ell'$.

<u>**Case `new`.**</u> We have

$$P, \textsc{bct}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t) : \ell$$

and

$$\frac{\overset{\text{(S-NEW)}}{P[pc^{\mathtt{C.m}}] = \mathtt{new\ D} \quad o \notin dom(H) \quad H' = H[o \mapsto (\rho^D, D)]}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, o \cdot s) \cdot \varphi, z \rangle_t}$$

By the configuration typing rules, we get

$$\Big(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i \qquad \ell'_i = \ell_i \ \& \ \widehat{\mathscr{T}'_i} \ \& \ \widecheck{\mathscr{R}'_i} \ \& \ \widehat{Z'_i}^t\Big)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}$$
$$P, \textsc{bct}, \Gamma'', \mathscr{P}'', \mathscr{T}'', \mathscr{R}'', K'' \vdash \langle \varphi, z \backslash Z'' \rangle_t : \ell_\varphi \qquad \mathscr{P}'' = \Big(\overline{(F'', S'')}, Z''\Big)$$
$$Z'_{pc^{\mathtt{C.m}}} = \overline{a'} \cdot a \qquad \overline{a'} \cdot Z'' \approx z \qquad S''_{pc^{\mathtt{C.m}}} \approx s \qquad F''_{pc^{\mathtt{C.m}}} \approx f$$

Therefore

$$\ell = (\sum_{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})} \ell'_i) + \ell_\varphi \tag{8}$$

From $(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}$ and Definition 6 we derive that

$$\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', pc^{\mathtt{C.m}} \vdash_t P[\mathtt{C.m}] : \ell_{pc^{\mathtt{C.m}}}$$

and

$$\big(\textsc{bct}, \Gamma', F', S', Z', \mathscr{T}', \mathscr{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i\big)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)}.$$

Therefore $\ell = \ell'_{pc^{\mathtt{C.m}}} + (\sum_{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)} \ell'_i) + \ell_\varphi$.

By these hypotheses, it is easy to show the existence of Γ''', $\mathscr{P}'$, $\mathscr{T}''$, $\mathscr{R}''$, K'' and ℓ'' such that $\Gamma''' \approx H'$ and $P, \textsc{bct}, \Gamma''', \mathscr{P}', \mathscr{T}'', \mathscr{R}'', K'' \vdash (\Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, o \cdot s) \cdot \varphi, z \rangle_t) : \ell'$. In particular, assuming $\mathsf{fields}(\mathtt{D}) = \overline{\mathtt{f}}$, it is sufficient to let $\Gamma''' = \Gamma'[o \mapsto ([\overline{\mathtt{f} : \top}], \mathtt{D})] \cdot \Gamma''$ and $S''_{pc^{\mathtt{C.m}}+1} = o \cdot S''_{pc^{\mathtt{C.m}}}$.

In this case, it is not evident that $\ell \succcurlyeq \ell'$. Actually, by the typing rule for `new`, the new name is a symbolic name returned by the function $names(pc^{\mathtt{C.m}})$, let it be a. Therefore, $\Gamma'_{pc^{\mathtt{C.m}}+1} = \Gamma'_{pc^{\mathtt{C.m}}}[a \mapsto ([\overline{\mathtt{f} : T}], \mathtt{D})]$, and $S'_{pc^{\mathtt{C.m}}+1} = a \cdot S''_{pc^{\mathtt{C.m}}}$. However, we cannot assume that $a = o$ because it may be already "in use" – this is the case when `C.m` is either recursive or $pc^{\mathtt{C.m}}$ is inside an iteration.

There are two cases: (i) `D` is not a subclass of `Thread` and (ii) `D` is a subclass of `Thread`. In case (i) it easy to verify that $\ell \succcurlyeq \ell'\{^a/_o\} \succcurlyeq \ell'$. In case (ii) we may have $\ell' = (b, c)_a \ \& \ (c, b)_o$.

When ℓ' has such a value, we have $\ell'\{^a/_o\} \not\succcurlyeq \ell'$ because the name o may appear as index of a dependency pair. In such situation, $(b,c)_a \;\&\; (c,b)_o$ is a circular dependency, while $\ell'\{^a/_o\} = (b,c)_a \;\&\; (c,b)_a$ is not – see Sections 5 and C.

To solve this criticality, whenever a new thread is created – either because a `start` is executed or because it is returned by a method invocation, *cf.* the sets $\mathscr{T}$ and $\mathscr{R}$ – we verify whether we are inside a recursion or an iteration, which are the two cases when $names(pc^{\texttt{C.m}})$ may return a name that already exists. In these two cases, the name a is added to the set $\mathscr{R}$ instead of $\mathscr{T}$. As a consequence, the lam ℓ' will contain terms $\texttt{RUN}((o\boxed{\texttt{f} : \rho}, \texttt{D}))$ while, correspondingly, ℓ will contain terms $\texttt{RUN}((o\boxed{\texttt{f} : \rho}, \texttt{D}))\{^a/_o\}$. However, since

$$\texttt{RUN}((o\boxed{\texttt{f} : \rho}, \texttt{D})) = \texttt{D.run}((o\boxed{\texttt{f} : \rho}, \texttt{D}), o, lock_o) \;\&\; (\nu\, o')\,\texttt{RUN}((o'\boxed{\texttt{f} : \rho}, \texttt{D}))\;,$$

replacing o in $\texttt{RUN}((o\boxed{\texttt{f} : \rho}, \texttt{D}))$ with any other object name does not change the circularity predicate (because the function is unfolded in a lam that has an invocation of the same function on a fresh name in parallel). Therefore, it is possible to derive $\ell\{^a/_t\} \succcurlyeq \ell$. ◂

C.4 Deadlocks and circularities

To conclude the proof of correctness we need to bridge the gap between the notion of deadlock in JVM and the notion of circularity in a lam program.

▸ **Definition 9.** A JVM configuration $\Vdash_H \langle\varphi_1, z_1\rangle_{t_1} \cdots \langle\varphi_n, z_n\rangle_{t_n}$ is *deadlocked* if there are $i_1, \cdots, i_k \in 1..n$ such that, for every $j \in \{i_1, \cdots, i_k\}$ one of the following holds

- $\varphi_j = (pc_j, f_j, s_j) \cdot \varphi'_j$ and $P[pc_j] = \texttt{monitorenter}$ and $s_j = a \cdot s'_j$ and $a \in z_h$ with $h \in \{i_1, \cdots, i_k\}\backslash j$;
- $\varphi_j = (pc_j, f_j, s_j) \cdot \varphi'_j$ and $P[pc_j] = \texttt{join}$ and $s_j = t_h \cdot s'_j$ with $h \in \{i_1, \cdots, i_k\}$.

The following statement is a straightforward consequence of the definitions.

▸ **Proposition 10.** *Let $\Vdash_H \mathscr{C}$ be deadlocked and let $P, \text{BCT}, \Gamma, F, S, Z, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \mathscr{C}) : \ell$. Then ℓ has a circularity.*

▸ **Theorem 11.** *Let P be a $\texttt{JVML}_d$ program and $\Gamma \approx H$, $F \approx_\Gamma f_\perp[0 \mapsto main]$, $S \approx_\Gamma \epsilon$, and $Z \approx_\Gamma \epsilon$. If*

1. $P, \text{BCT}, \Gamma, \mathscr{P}, \mathscr{T}, \mathscr{R}, K \vdash (\Vdash_H \langle(1^{\texttt{C}.main}, f_\perp[0 \mapsto main], \epsilon), \epsilon\rangle_{main}) : \ell$
2. *and* $\Vdash_H \langle(1^{\texttt{C}.main}, f_\perp[0 \mapsto main], \epsilon), \epsilon\rangle_{main} \rightarrow^* \;\Vdash_{H'} \mathscr{C}'$
3. *and* $\Vdash_{H'} \mathscr{C}'$ *is deadlocked*

then ℓ has a circularity.

Proof. The proof is an immediate consequence of the previous results. By 1 and 2 and Theorem 8 we have the existence of ℓ' and Γ', $\mathscr{P}'$, $\mathscr{T}'$,$\mathscr{R}'$,K' such that $\Gamma' \approx H'$ and $P, \text{BCT}, \Gamma', \mathscr{P}', \mathscr{T}', \mathscr{R}', K' \vdash (\Vdash_{H'} \mathscr{C}') : \ell'$ and $\ell \succcurlyeq \ell'$. By 3 and Proposition 10, we have that ℓ' has a circularity. By $\ell \succcurlyeq \ell'$ and Lemma 10, we have that ℓ has a circularity as well. ◂