A Compositional Deadlock Detector for Android Java

> 作者：James Brotherston, Jens Palsberg 等
>
> 单位：University College London, Facebook UK

> 会议：ASE 21

>
> 链接：http://www0.cs.ucl.ac.uk/staff/J.Brotherston/ASE21/deadlocks_final.pdf

Abstract

目标

- 为FaceBook数千万的Android Java应用开发静态死锁分析
- 在代码审查阶段，只对修改后的代码及其引用做分析，以提高效率
- 15min内完成分析

首先将真实语言建模为**具有平衡的可重入锁、非确定性的迭代和分支以及非递归过程调用的**抽象语言。作者证明了在这种抽象语言中，死锁的存在等同于每个程序线程的 critical pairs 集合的某个条件；这些 critical pairs 记录了对于线程的在获得新锁时，哪些锁是被持有的所有可能情况。由于 critical pairs 是有限的，所以死锁检测模型是NP且可解的。

基于以上实现Java代码死锁分析，该实现的核心是一种算法，它以组合式的抽象解释方式计算 critical pairs ，在准指数时间内运行。

1 Introduction

在一个并发程序中，死锁描述了这样一种情况：对于该程序的某些线程子集来说，任何线程都不可能最终执行其下一个命令。

目标：

1. 快，15min内完成分析
2. 报告有用，false positive少

第一步，将真实语言建模为**具有平衡的可重入锁、非确定性的迭代和分支以及非递归过程调用的**抽象语言。这是Java的一个 over-approximate 模型，所有关于变量和内存分配的信息都被抽象出来。

在抽象程序中，死锁的存在被描述为线程的 critical pairs 的条件。

比如线程的一对 critical pair 是 (X, l) 含义是线程获得了一个未持有的锁 l，同时已经持有锁的集合 X 。

对于两个线程(C_1, C_2) ，当且仅当 critical pairs (X_1, l_1) 和 (X_2, l_2) 分别使得$l_1 \in X_2$ 、$l_2 \in X_1$ 和 $X_1 \cap X_2 = \phi$，才存在死锁。

上述定义正确性取决于：在我们的语言中，锁是平衡的，即任何线程必须按照获得锁的相反顺序释放锁，"后进先出"，类似于栈。Java语言的synchronized和C++的std::lock_guard都是如此定义的。

Example 1.1. 考虑一个双线程程序$C_1 \mid\mid C_2$，其中C_1 和 C_2获得锁 (acq(-)) 和释放锁 (rel(-)) 的顺序是相反的。

$$
\begin{aligned}
C_1: &\quad \texttt{acq}(x); \texttt{acq}(y); \texttt{skip}; \texttt{rel}(y); \texttt{rel}(x) \\
C_2: &\quad \texttt{acq}(y); \texttt{acq}(x); \texttt{skip}; \texttt{rel}(x); \texttt{rel}(y)
\end{aligned}
$$

C_1 有这么两对 (ϕ, x)和$(\{x\}, y)$ critical pairs

C_2 有这么两对 (ϕ, y)和$(\{y\}, x)$ critical pairs

有 $(X_1, l_1) = (\{x\}, y)$ 和 $(X_2, l_2) = (\{y\}, x)$ 所以有死锁。

如果把程序改成 $C_1 \mid\mid C_2$，其中$C_1' = acq(z); C_1; rel(z)$ 和 $C_2' = acq(z); C_2; rel(z)$

C_1' 有这么三对 (ϕ, z)、$(\{z\}, x)$、$(\{z, x\}, y)$ critical pairs

C_2' 有这么三对 (ϕ, z)、$(\{z\}, y)$、$(\{z, y\}, x)$ critical pairs

不符合上述条件 不会死锁。

基于这些理论做死锁分析，上下文不敏感的程序分析，以抽象解释的方式计算 critical pairs。死锁分析器叫 `starvation` INFER的一部分

starvation和其他工具的两个不同：

- 其他工具大多需要整个程序才能运行，starvation专注于分析代码更改
- 更优先complete而不是sound

> 还不是 over-approximated...

2 Program Syntax and Semantics

Syntax

Syntax. We let Locks be a finite set of *global lock names* and Procs a set of *procedure names*. We define *statements* C as follows, where ℓ ranges over Locks and p over Procs:

$$\begin{aligned} C := \ & \texttt{skip} \mid p() \mid \texttt{acq}(\ell) \mid \texttt{rel}(\ell) \mid C; C \\ & \mid \texttt{if}(*)\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}(*)\ \texttt{do}\ C \end{aligned}$$

function body(): Procs → Stmt **过程名到语句的映射**

function callees(·): Stmt → P(Procs) **语句到过程名的映射**

We forbid recursion in statements; i.e., for all $p \in Procs, p \notin callees(body(p))$. (有点没懂

通过以下语法定义 balanced，保证上述后进先出的成对操作。

$$C := \texttt{skip} \mid p() \mid \texttt{acq}(\ell); C; \texttt{rel}(\ell) \mid C; C$$
$$\mid \texttt{if}(*) \texttt{ then } C \texttt{ else } C \mid \texttt{while}(*) \texttt{ do } C$$

平衡语句必须只调用平衡过程：if C is balanced and p ∈ callees(C), then body(p) must be balanced as well.

Semantics

只记录和锁相关的信息，可重入。

锁状态定义成 function L: Locks → $\mathbb{N}$ **记录每个锁被获取的次数**

notation $\lfloor L \rfloor$**为** $\{l \in Locks | L(l) > 0\}$ **被获取的所有锁元素集合**

notation $L_1 \# L_2$**为** $\lfloor L_1 \rfloor \cap \lfloor L_2 \rfloor = \phi$ write ∅ for the lock state sending all locks to 0

notation $L[l++](l) = L(l) + 1$ $\quad$ $L[l--](l) = L(l) - 1$

配置定义成如下的pair $< C, L >$其中C是语句，L是锁状态。

对于并行程序来说记作 $< C_1||\ldots||C_n, (L_1, \ldots, L_n) >$ 或 $< C_1，\ L_1 > ||\ldots|| < C_n, L_n >$ 或 $||_{1\leq i\leq n} < C_i, L_i >$

$< C_i，\ L_i > \# < C_j, L_j >== L_i \# L_j$

> n 线程数

普通程序的小步语义$\rightarrow$，并行程序的小步语义$\rightsquigarrow$

$$\langle \texttt{skip}; C, L\rangle \rightarrow \langle C, L\rangle \quad (\texttt{skip}) \qquad \langle \texttt{if}(*)\ \texttt{then}\ C_a\ \texttt{else}\ C_b, L\rangle \rightarrow \langle C_a, L\rangle \quad (\texttt{if1})$$

$$\langle p(), L\rangle \rightarrow \langle \texttt{body}(p), L\rangle \quad (\texttt{proc}) \qquad \langle \texttt{if}(*)\ \texttt{then}\ C_a\ \texttt{else}\ C_b, L\rangle \rightarrow \langle C_b, L\rangle \quad (\texttt{if2})$$

$$\langle \texttt{acq}(\ell), L\rangle \rightarrow \langle \texttt{skip}, L[\ell{+}{+}]\rangle \quad (\texttt{acq}) \qquad \langle \texttt{while}(*)\ \texttt{do}\ C, L\rangle \rightarrow \langle \texttt{skip}, L\rangle \quad (\texttt{while1})$$

$$\langle \texttt{rel}(\ell), L\rangle \rightarrow \langle \texttt{skip}, L[\ell{-}{-}]\rangle\ (L(\ell) > 0) \quad (\texttt{rel}) \qquad \langle \texttt{while}(*)\ \texttt{do}\ C, L\rangle \rightarrow \langle C; \texttt{while}(*)\ \texttt{do}\ C, L\rangle \quad (\texttt{while2})$$

$$\frac{\langle C_1, L\rangle \rightarrow \langle C_1', L'\rangle}{\langle C_1; C_2, L\rangle \rightarrow \langle C_1'; C_2, L'\rangle}\ (\texttt{seq}) \qquad \frac{\langle C_i, L_i\rangle \rightarrow \langle C_i', L_i'\rangle \qquad L_i' \,\#\, \{L_j \mid j \neq i\}}{\langle C_1 \,||\ldots||\, C_n, (L_1, \ldots, L_n)\rangle \rightsquigarrow \langle C_1 \,||\ldots||\, C_i' \,||\ldots||\, C_n, (L_1, \ldots, L_i', \ldots, L_n)\rangle}\ (\texttt{par}\ i)$$

Fig. 1. Small-step semantics for statements ($\rightarrow$) and parallel programs ($\rightsquigarrow$).

通过$\rightarrow$的自反传递闭包$\rightarrow^*$表示执行，通过$\rightsquigarrow$的自反传递闭包$\rightsquigarrow^*$表示执行

Example：

对于执行 $(\gamma_i)_{i\geq 0}$ 用符号 $\gamma_0 \rightarrow^* \gamma_n$ 表示。（一个执行可以看成是上面的一个pair，

Remark 2.1. *For any concurrent execution* $\gamma_1 \,||\, \ldots \,||\, \gamma_n \rightsquigarrow^* \gamma'_1 \,||\, \ldots \,||\, \gamma'_n$ *there exist standard executions* $\gamma_i \rightarrow^* \gamma'_i$ *for each* $1 \leq i \leq n$. *Furthermore, if* $\gamma_i \# \gamma_j$, *then* $\gamma'_i \# \gamma'_j$; *i.e., no two threads can acquire the same lock simultaneously.*

> (如果已经有了一个并发执行，那么肯定没有死锁

定义程序的死锁为至少有俩线程死锁，引入了一种将并发配置投影到线程子集上的符号：

$\sigma =< C_1||\ldots||C_n, (L_1, \ldots, L_n) >$ 是一个并发配置，
$I = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ 是一组线程索引， σ_I 是并发配置
$< C_{i_1}, \ L_{i_1} > ||\ldots|| < C_{i_m}, L_{i_m} >$。

线程死锁定义

Definition 2.2 (Deadlock). *A concurrent configuration, say* $\sigma = \langle C_1 \mid\mid \ldots \mid\mid C_n, (L_1, \ldots, L_n)\rangle$, *is* deadlocked *if there is a sequential transition* $\langle C_i, L_i\rangle \rightarrow \langle C_i', L_i'\rangle$ *for each thread (i.e. for all* $1 \leq i \leq n$*) but there is no concurrent transition* $\sigma \rightsquigarrow \sigma'$.

有一个并发配置，而且单独的每一步可以转移，但是没有整个的并发转移

程序死锁定义，存在一个死锁的线程子集。

一个线程集合是死锁的，那么他的超集也是死锁的。

Proposition 2.3. *Let* $\sigma = \langle C_1 \mid\mid \ldots \mid\mid C_n, (L_1, \ldots, L_n)\rangle$ *be a concurrent configuration such that* $L_i \# L_j$ *for all* $i \neq j$*. The configuration* σ *is deadlocked iff there are statements* $D_1, \ldots, D_n$ *and locks* $\ell_1, \ldots, \ell_n$ *such that, for all* $1 \leq i \leq n$

$$\langle C_i, L_i\rangle \to \langle D_i, L_i[\ell_i{+}{+}]\rangle \quad \textit{and} \quad \ell_i \in \bigcup_{j\neq i} \lfloor L_j \rfloor \ .$$

给出了从不死锁到死锁的转移定义

3 EXECUTIONS AND TRACES

这一部分给出 **语句执行** 到 **锁获取和释放** 的抽象定义—— "execution's trace"

Example：

$$\begin{aligned}&\mathtt{acq}(\ell);\mathtt{if}(*)\ \mathtt{then}\ (\mathtt{acq}(j);\mathtt{skip};\mathtt{rel}(j))\\&\qquad\qquad\mathtt{else}\ (\mathtt{acq}(k);\mathtt{skip};\mathtt{rel}(k));\mathtt{rel}(\ell)\end{aligned}$$

上述程序有两条可能的traces $l\ j\ \overline{j}\ \overline{l}$ 和 $l\ k\ \overline{k}\ \overline{l}$ 。上述traces保留了所有对锁有影响的操作。

> 这一章就是在形式化的讲为啥上述example的trace是这样的

Definition 3.1. *The* lock alphabet Σ *is defined as the union of two disjoint copies of* Locks*:*

$$\Sigma := \{\ell \mid \ell \in \mathsf{Locks}\} \cup \{\overline{\ell} \mid \overline{\ell} \in \mathsf{Locks}\} \ .$$

锁表 Σ 定义，应该就是一个包含所有锁相关操作的集合

> 跟前文过渡不自然

准锁定状态是 Locks → $\mathbb{Z}$ 的一个函数？（Z没说是啥，盲猜整数，但是为啥前面是N

从锁定状态到准锁定状态的符号：$[l++]$和$[l--]$，同时定义准锁定状态上的 +，表示函数的**逐点和**，即 $(f+g)(x)=f(x)+g(x)$。

定义 Σ-words 到 **准锁定状态** 的函数 $\langle\cdot\rangle$

$$\langle\epsilon\rangle := \phi \quad \langle u\ l\rangle := \langle u\rangle[l++] \quad \langle u\ \overline{l}\rangle := \langle u\rangle[l--]$$

根据上述逐点和定义有 **Lemma 3.2.** For all $u, v \in \Sigma^*$，有$\langle u\ v\rangle = \langle u\rangle + \langle v\rangle$

> 类似前文锁定状态函数L
> 然后后面的内容好像又跟准锁定状态无关了？？？

对于一步执行，要么一个锁被释放，要么一个锁被获取，要么没有锁相关操作。形式化定义：

Definition 3.3. *Given a transition* $\langle C, L\rangle \to \langle C', L'\rangle$, *we define its* trace $u \in \Sigma \cup \{\varepsilon\}$ *as follows:*

$$u = \begin{cases} \varepsilon & \text{if } L' = L \\ \ell & \text{if } L' = L[\ell{+}{+}] \\ \overline{\ell} & \text{if } L' = L[\ell{-}{-}]. \end{cases}$$

注：一个转换（箭头）或者执行（自反传递闭包）的trace通常写在箭头上面，

Proposition 3.4. *For any execution* $\langle C_0, L_0\rangle \xrightarrow{u}{}^* \langle C_n, L_n\rangle$ *and statement* C*, we can obtain an execution* $\langle C_0; C, L_0\rangle \xrightarrow{u}{}^* \langle C_n; C, L_n\rangle$ *with the same trace.*

上述命题可以直接用语义规则归纳证明。直观理解的话，加一句不影响执行的语句，不影响trace。

定义语句的形式化语言，也可以理解成一组可能的执行traces：

Definition 3.5. *The* language $\mathcal{L}(C)$ *of a statement* C *is defined inductively as follows:*

$$\begin{aligned}
\mathcal{L}(\texttt{skip}) &:= \{\varepsilon\} & \mathcal{L}(p()) &:= \mathcal{L}(\texttt{body}(p)) \\
\mathcal{L}(\texttt{acq}(\ell)) &:= \{\ell\} & \mathcal{L}(\texttt{rel}(\ell)) &:= \{\bar{\ell}\} \\
\mathcal{L}(C_1; C_2) &:= \mathcal{L}(C_1) \cdot \mathcal{L}(C_2) \\
\mathcal{L}(\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2) &:= \mathcal{L}(C_1) \cup \mathcal{L}(C_2) \\
\mathcal{L}(\texttt{while}(*)\ \texttt{do}\ C) &:= \mathcal{L}(C)^\star
\end{aligned}$$

Remark 3.6. $\mathcal{L}(C)$ *is in fact a* regular language *over* Σ.

> 我理解和前述语义和语法的区别是这里是抽象语言锁集合上的抽象语言，只有锁相关的语义
> 后文就开始讲抽象语言上的执行定义和对锁状态的影响

如下引理确立了执行的效果本质上由其traces决定。

Lemma 3.7. *For any execution* $\pi : \langle C, L\rangle \xrightarrow{u}{}^{*} \langle C', L'\rangle$ *we have* $L' = L + \langle u\rangle$ *and* $u \cdot \mathcal{L}(C') \subseteq \mathcal{L}(C)$.

Proof. We first prove the lemma for a single $\rightarrow$-step by rule induction on the semantics. The general result then follows by reflexive-transitive induction on $\rightarrow^{*}$. □

> 没特别理解证明，有种看的《Formal Reasoning About Programs》感觉，直观理解就是后面的分别给出了一个前述执行对于锁状态和语句的影响描述
> 不过上面最后一个式子为啥是子集

给了一个Dyck words D的形式化定义，类似左括号数量大于右括号，且总数量相等。

Definition 3.8. *The language $\mathcal{D}$ of* Dyck words *over Σ is generated by the following grammar:*

$$D := \varepsilon \mid D\,D \mid \ell\,D\,\bar{\ell}.$$

一些Dyck相关性质：

Lemma 3.11. 对于任意 $u \in D$ 有 $\langle u \rangle = \phi$

> write ∅ for the lock state sending all locks to 0

还有一个没理解的

Lemma 3.12. 对于任意 $u\ v \in D$ 有 $\langle u \rangle \in \text{Locks} \to \backslash\mathbf{N}$

> 这边又莫名其妙变成 $\to \backslash\mathbf{N}$ 了

为平衡语句的执行建模，对于后面分析并发执行非常重要。

Lemma 3.13. *Let C be a balanced statement. For any execution of the form*

$$\langle C, \varnothing \rangle = \langle C_0, L_0 \rangle \to^* \langle C_n, L_n \rangle \to \langle C_{n+1}, L_n[\ell{--}] \rangle \ ,$$

there exists $j < n$ such that $L_j = L_n[\ell{--}]$ and $L_{j+1} = L_n$.

> 其实前面就没太理解 transition 和 execution 的区别，不知道为啥这里记号不一样

Lemma 3.14. *Let C be a balanced statement, and let $u\,v \in \mathcal{L}(C)$. For any lock state L, there is a statement D and an execution $\pi : \langle C, L\rangle \xrightarrow{u}{}^{*} \langle D, L + \langle u\rangle\rangle$ such that $v \in \mathcal{L}(D)$. If $v = \varepsilon$, then this statement also holds when $D = \texttt{skip}$.*

> 跟Lemma 3.7类似

Corollary 3.15. *For any balanced statement C, we have*

$$\mathcal{L}(C) = \{u \mid \langle C, \varnothing \rangle \xrightarrow{u}{}^{*} \langle \texttt{skip}, \varnothing \rangle\} \ .$$

> 拿Lemma 3.7. 和3.14.结合一下

所以对于如下语句集合

$$\begin{aligned}&\texttt{acq}(\ell);\texttt{if}(*)\ \texttt{then}\ (\texttt{acq}(j);\texttt{skip};\texttt{rel}(j))\\&\qquad\qquad\texttt{else}\ (\texttt{acq}(k);\texttt{skip};\texttt{rel}(k));\texttt{rel}(\ell)\end{aligned}$$

有如下两条traces，$L(C)=\{l\ j\ \overline{j}\ \overline{l}, l\ k\ \overline{k}\ \overline{l}\}$

$$\langle C,\varnothing\rangle \xrightarrow{\ell} \langle \_,\{\ell\}\rangle \xrightarrow{\varepsilon} \langle \_,\{\ell\}\rangle \xrightarrow{j} \langle \_,\{\ell,j\}\rangle \xrightarrow{\overline{j}} \langle \_,\{\ell\}\rangle \xrightarrow{\overline{\ell}} \langle \texttt{skip},\varnothing\rangle$$

$$\langle C,\varnothing\rangle \xrightarrow{\ell} \langle \_,\{\ell\}\rangle \xrightarrow{\varepsilon} \langle \_,\{\ell\}\rangle \xrightarrow{k} \langle \_,\{\ell,k\}\rangle \xrightarrow{\overline{k}} \langle \_,\{\ell\}\rangle \xrightarrow{\overline{\ell}} \langle \texttt{skip},\varnothing\rangle$$

> call back前面

4 CHARACTERISATION OF DEADLOCK EXISTENCE

结论：基于Coq证明死锁存在相当于是线程的摘要（关键对集合）存在某种冲突。

同时说明在一个奇怪的条件下 sound and complete

没啥意义，姚老师说就是作者复用之前的工作

5 COMPUTING CRITICAL PAIRS

在定理4.4中介绍了并行程序的死锁规约到抽象关键对的条件。

本章证明为什么关键对对于平衡语句C是可计算的，且是NP的（NP也算可计算？？

6 IMPLEMENTATION AND IMPACT

6.1 关键对的抽象解释计算

6.2 讲怎么分析Android更改

6.3 讲开销

我觉得这一章最关键。

6.1 Core analysis in abstract interpretation style

实现：以抽象解释方式计算语句的关键对。

对于语句C，定义了抽象状态的分析函数 $\llbracket C \rrbracket(\cdot)$ ，用于跟踪**锁状态**和在C**执行过程中产生的关键对**。

Definition 6.1. *An* abstract state *is a pair* $\langle L, Z \rangle$*, where* L *is a lock state and* $Z \subseteq 2^{\mathsf{Locks}} \times \mathsf{Locks}$*. We define a partial* join *operation* $\sqcup$ *on abstract states by* $\langle L, Z_1 \rangle \sqcup \langle L, Z_2 \rangle = \langle L, Z_1 \cup Z_2 \rangle$*. We often write* α *for abstract states, and* $\alpha_\perp$ *for the "empty" abstract state* $\langle \varnothing, \emptyset \rangle$*.*

> L说是锁状态按照前述定义**记录每个锁被获取的次数**，Z没说我理解应该是 critical pair 的集合。

$$
\begin{aligned}
[\![\texttt{acq}(\ell)]\!]\langle L, Z\rangle &= \langle L[\ell{+}{+}], Z \cup Z'\rangle \\
\text{where } Z' &= \begin{cases} \{(\lfloor L \rfloor, \ell)\} & \text{if } L(\ell) = 0 \\ \emptyset & \text{if } L(\ell) > 0 \end{cases} \\
[\![\texttt{rel}(\ell)]\!]\langle L, Z\rangle &= \langle L[\ell{-}{-}], Z\rangle \\
[\![p()]\!]\langle L, Z\rangle &= \langle L, Z \cup Z'\rangle \\
\text{where } \langle \_, Z''\rangle &= [\![\mathsf{body}(p)]\!]\alpha_\perp \text{ in} \\
Z' &= \{(\lfloor L \rfloor \cup M, \ell) \mid (M, \ell) \in Z'' \wedge L(\ell) = 0\} \\
[\![\texttt{skip}]\!]\alpha &= \alpha \\
[\![C_1; C_2]\!]\alpha &= [\![C_2]\!]([\![C_1]\!]\alpha) \\
[\![\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\alpha &= ([\![C_1]\!]\alpha) \sqcup ([\![C_2]\!]\alpha) \\
[\![\texttt{while}(*)\ \texttt{do}\ C]\!]\alpha &= \bigsqcup_{n=0}^{\infty} [\![C]\!]^n \alpha
\end{aligned}
$$

Fig. 2. Abstract analysis definition.

一些语句的抽象解释含义

注意：计算过程调用 p() 的**关键对** 仅取决于**当前抽象状态** $\langle L, Z \rangle$ 和 处于空状态的**过程主体**的关键对$\llbracket body(p) \rrbracket \alpha_{\perp}$，所以可以先预计算，然后部分变化时只计算相关从属。

> 为啥？？？不过跟我们也不是很相关，没有这么高的代码量要求。

Proposition 6.2. *For any balanced statement C and abstract state $\alpha = \langle L, Z \rangle$, the result $[\![C]\!]\alpha$ of the analysis is given by*

$$\langle L, Z \cup \{(\lfloor L \rfloor \cup X, \ell) \mid (X, \ell) \in \mathbf{Crit}(C) \text{ and } L(\ell) = 0\} \rangle \ .$$

Thus $[\![C]\!]\alpha_\perp = \langle \varnothing, \mathbf{Crit}(C) \rangle$. Moreover, $[\![C]\!]\alpha$ is computable.

Proof. By structural induction on C, making use of the equations (C1)–(C6) in Proposition 5.4. □

> 对于任意 **平衡语句C** 和 **抽象状态** $\alpha = \langle L, Z \rangle$，$\llbracket C \rrbracket \alpha$ 的结果只取决于上述很长的公式。
> 大概理解因为平衡语句，所以L不变，后面那个加入如后面所示的新的 critical pair。

Example 6.3. Continuing Example 5.2, statement C is:

$$\begin{aligned}&\texttt{acq}(\ell);\texttt{if}(*)\ \texttt{then}\ (\texttt{acq}(j);\texttt{skip};\texttt{rel}(j))\\&\qquad\qquad\texttt{else}\ (\texttt{acq}(k);\texttt{skip};\texttt{rel}(k));\texttt{rel}(\ell)\ .\end{aligned}$$

再结合上述图2 If else，这个例子被形式化为

$$\begin{aligned}[\![C]\!]\alpha_\perp &= [\![\texttt{acq}(\ell);C';\texttt{rel}(\ell)]\!]\alpha_\perp\\&= [\![C';\texttt{rel}(\ell)]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp\\&= [\![\texttt{rel}(\ell)]\!][\![C']\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp\\&= [\![\texttt{rel}(\ell)]\!][\![\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp\\&= [\![\texttt{rel}(\ell)]\!]([\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp \sqcup [\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp)\ .\end{aligned}$$

再对于每个语句分别抽象解释

比如

$$\llbracket acq(l) \rrbracket \alpha_{\perp} = \llbracket acq(l) \rrbracket \langle \Phi, \phi \rangle = \langle \Phi[l++], \{(\phi, l)\} \rangle$$

$$\begin{aligned} [\![\mathsf{acq}(\ell)]\!]\alpha_{\perp} &= [\![\mathsf{acq}(\ell)]\!]\langle \varnothing, \emptyset \rangle \\ &= \langle \varnothing[\ell{+}{+}], \{(\emptyset, \ell)\} \rangle \ . \end{aligned}$$

所以有

We write L_ℓ for the lock state $\varnothing[\ell{+}{+}]$ sending lock ℓ to 1 (and all other locks to 0). Next, we have

$$
\begin{aligned}
&[\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp \\
&= [\![C_1]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle \\
&= [\![\texttt{acq}(j); \texttt{skip}; \texttt{rel}(j)]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle \\
&= [\![\texttt{skip}; \texttt{rel}(j)]\!][\![\texttt{acq}(j)]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle \\
&= [\![\texttt{rel}(j)]\!][\![\texttt{skip}]\!][\![\texttt{acq}(j)]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle \\
&= [\![\texttt{rel}(j)]\!][\![\texttt{skip}]\!]\langle L_\ell[j{+}{+}], \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle \\
&= [\![\texttt{rel}(j)]\!]\langle L_\ell[j{+}{+}], \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle \\
&= \langle L_\ell[j{+}{+}][j{-}{-}], \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle \\
&= \langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle \ .
\end{aligned}
$$

And, by a similar calculation,

$$[\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp = \langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, k)\}\rangle \ .$$

继续推导如下，就完成了对于critical pair的计算

Now, using the definition of our abstract join $\sqcup$, we have

$$\begin{aligned}
&[\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp \sqcup [\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp \\
&= \langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle \sqcup \langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, k)\}\rangle \\
&= \langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, j), (\{\ell\}, k)\}\rangle \ .
\end{aligned}$$

Thus, putting everything together, we get

$$\begin{aligned}
[\![C]\!]\alpha_\perp &= [\![\texttt{rel}(\ell)]\!]([\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp \sqcup [\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp) \\
&= [\![\texttt{rel}(\ell)]\!]\langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, j), (\{\ell\}, k)\}\rangle \\
&= \langle L_\ell[\ell{--}], \{(\emptyset, \ell), (\{\ell\}, j), (\{\ell\}, k)\}\rangle \\
&= \langle \varnothing, \{(\emptyset, \ell), (\{\ell\}, j), (\{\ell\}, k)\}\rangle \ .
\end{aligned}$$

6.2 Analysing Android Java code changes

一些理论和实现的差异 和 Android Java特性（paragraphs on Non-deterministic control and Concurrency inference）

Balanced locking

分析的正确性依赖于此。一般都是平衡的，Java通过synchronized提供支持，（Apache SSHD没有不平衡的）。所以Infer基于平衡锁建模。

Non-deterministic control.

Java中的控制大多是确定性的，所以抽象语义**过近似**了。（应该是控制流

对于只有非确定性控制的是sound and complete的。

误报原因：两个条件的不敏感

1. 锁获取是否成功，比如Lock.trylock
2. 当前线程是否是UI线程

解决办法：引入部分路径敏感性，消除一些误报。

Lock names

抽象集合 Locks 必须近似于可作为 monitor 的 Java对象集合，。

他们没有基于指针分析实现，因为通常指针分析只能基于全程序实现，就不能做成组成式的分析了。

而是基于 access path（访问路径）：用程序变量根构建的语法表达式 和 field- 或 array-dereferencing 的迭代。比如，this.f.g表示 通过解引用字段f 和 对象this 的对象访问

> 原文：syntactic expressions built with a program variable root and iteration of field- or array-dereferencing
> 大概就是基于表达式做的一个静态分析。
> monitor大概就是我们需要关注的那些元素集合

将对象分类为**全局引用的**、**通过方法参数引用的**、**通过局部变量引用的**三类。

忽略通过局部变量引用的对象；对于全局变量引用的对象保持图2的调用规则。

对于通过参数引用的对象，在应用过程调用规则之前，我们将参数表达式替换为被调用方摘要上的参数。

比如，如果方法foo(x) 的 summary 包含锁 `x.f`，那么在foo(h.g)上应用过程调用规则将导致替换 `[h.g/x]`，并且在调用点生成的 critical pair 将涉及 monitor `h.g.f`。

> 一些用于过程间分析的差别
> 没有解释什么是 callee summary，论文中这段是一个出现summary的地方
> 我理解这个替换就是相当于把参数传进来

6.3 Industrial deployment and impact

3k行代码、流敏感、上下文不敏感、CI集成

两年部署、500份报告、54%修复。

46%：新的commit里已经修了、懒得处理（概率低）、false postive。

运行时间快平均90s for 5k methods。

7 Conclusions and Future Work

该抽象语言基于 平衡（嵌套）可重入锁，非确定性控制流命令，非递归过程调用。

其他比如变量赋值的特征都被抽象了，必要的。

在保留死锁存在的可解性的同时，进行扩展。

如果允许递归调用，会有很大问题。

1. 创造出无限不平衡的traces，比如 `p: acq(l); p(); rel(l);`
2. 不能直接通过对语句结构的归纳来推理。

如果允许控制流是确定性的，例如允许守护者查询锁状态，也同样存在问题，因为语句的关键对取决于它被执行的锁状态，这意味着至少我们需要一个更精细的抽象，以避免误报。

Allowing control flow to be deterministic, e.g. by allowing guards to query the lock state, is similarly problematic since the critical pairs of a statement are then dependent on the lock state in which it is executed, meaning that at the very least we would require a finer abstraction in order to avoid false positives.

> nondeterministic，可能在theorem5.5。
> 非确定性控制流和流敏感差别在哪?

Related Code

3k 行

https://github.com/facebook/infer/blob/main/infer/src/concurrency/starvation.ml

https://github.com/facebook/infer/blob/main/infer/src/concurrency/StarvationModels.ml

https://github.com/facebook/infer/blob/main/infer/src/concurrency/starvationDomain.ml

Thank You