Data races in Java and static analysis

Frédéric Dabrowski

[1]INRIA, LANDE

Outline

1. Concurrency in Java

2. Static Detection of dataraces
   - lock-based typing $\begin{cases} \text{Flanagan, Abadi \& Freund} \\ \text{Boyapati, Lee \& Rinard} \end{cases}$
   - Points-to analysis : Naik & Aiken
     (Points-to analysis + Type and effect system)

3. Conclusion and ongoing work

Concurrency in Java

Concurrency in Java

Concurrency model

- *Thread-based* concurrency : shared memory (fields of shared objects)
- lexically scoped *locking* construct : `synchronized`$(x)\{\ldots\}$
- *Preemptive* scheduling (Interleaving semantics)

Concurrency in Java

Interleaving semantics

(small step) sequential semantics	interleaving semantics
$t, Mem \rightarrow_{seq} t', Mem'$	$\dfrac{t_i, Mem \rightarrow_{seq} t'_i, Mem'}{\{\ldots, t_i, \ldots\}, Mem \rightarrow_{inter} \{\ldots, t'_i, \ldots\}, Mem'}$

Problem : This semantics is incomplete with respect to the Java Memory Model, **unless you write well-synchronized programs**

Concurrency in Java

Natural hypothesis : sequential consistency

Intuivively, **sequential consistency** means that all executions respect the program order.

$$\begin{array}{l} \texttt{void}\ mn()\{ \\ \quad a \qquad a \text{ should not observe } b \\ \quad \ldots \\ \quad b \\ \} \end{array}$$

Problem : enforcing sequential consistency for all Java programs makes many of the compiler/processor optimizations illegal.

Why ? some optimizations assume well-synchronized programs !

Concurrency in Java

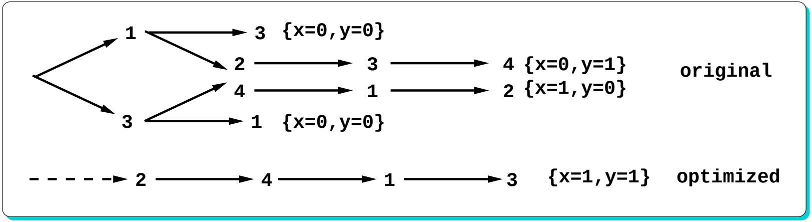
Example : code reordering (cache mechanisms,...)

Original code

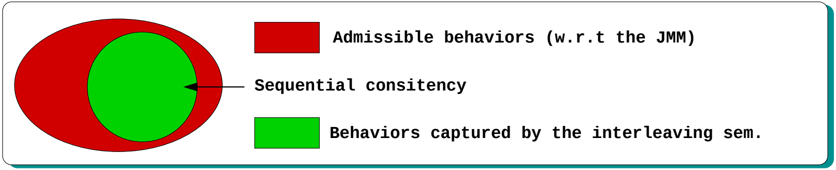
$$C.f = C.g = 0$$
$$\begin{array}{l|l} 1 : x = C.g; & 3 : y = C.f; \\ 2 : C.f = 1; & 4 : C.g = 1; \end{array}$$
$$\{Perm(1,2,3,4) \mid 1 < 2,\ 3 < 4\}$$

Optimized code

$$C.f = C.g = 0$$
$$\begin{array}{l|l} 2 : C.f = 1; & 4 : C.g = 1; \\ 1 : x = C.g; & 3 : y = C.f; \end{array}$$
$$\{Perm(1,2,3,4) \mid 2 < 1,\ 4 < 3\}$$



Concurrency in Java



Java's memory model is weak memory model

All executions of well-synchronized programs are sequentially consistent [a].

[a]Manson, Pugh & Adve : The Java Memory Model (Special Popl issue)

Programs must be well-synchronyzed

several static analysis depend on it

Concurrency in Java

Well-synchronized programs

(P_1) : For all execution (w.r.t the interleaving semantics), every conflicting actions a and b are synchronized

$$\texttt{synchronized}(C)\{(1 : x = C.g); (2 : C.f = 1)\} \;\Big|\; \texttt{synchronized}(C)\{(3 : y = C.f); (4 : C.g = 1)\}$$

lock(C) 1:x=C.g $\boxed{\text{2:C.f=1}}$ unlock(C) lock(C) $\boxed{\text{3:y=C.f}}$ 4:C.y=1 unlock(C)

seq locking seq

lock(C) $\boxed{\text{3:y=C.f}}$ 4:C.y=1 unlock(C) lock(C) 1:x=C.g $\boxed{\text{2:C.f=1}}$ unlock(C)

seq seq locking seq seq

compiler/jvm/jit : $(P_1) \Rightarrow$ every exec. is captured by the inter. sem.

Concurrency in Java

Happens-before relation

$\prec_{hb}$ is the transitive closure of the following rules :

- **sequentiality** $a^t \prec^1_{hb} b^t$
- **start/join synchronisation** $\begin{cases} t.\texttt{start}() \prec^1_{hb} a^t \\ a^t \prec^1_{hb} t.\texttt{join}() \end{cases}$
- **lock-based synchronisation**

$$\begin{cases} \texttt{unlock}(m) \prec^1_{hb} \texttt{lock}(m) \\ \texttt{write}(x.f) \prec^1_{hb} \texttt{read}(x.f) \qquad (f \text{ volatile}) \end{cases}$$

Concurrency in Java

Data races

definition (JMM) : a program has a data race is there exists an execution with two conflicting actions not not ordered by $\prec_{hb}$.

alternative definition : a program has a data race if there exists an execution such that, at some point, there is a non deterministic choice (interleaving semantics) among two conflicting actions.

Problem (undecidable) :

Given a program, are all executions of that program race free ?

Static detection of data races

RCC Java

[PLDI'00] Type-based race detection for Java (Flanagan and Freund)

- supports classes parameterized by locks of given types (Dependent types)
- introduce a notion of thread local classes
- Fields protected by locks (static fields)
- Encapsulation : self-protected class
- Extend previous work based on a simple thread calculus

Example

```
class A⟨ghost Object x⟩{
    Object y = new Object() guarded_by x;
     void set(Object z) requires x{
        this.y = z;
    }
}

class B{
    final Object z = new Object();
    A⟨this.z⟩ x = new A⟨this.z⟩();
    void f(){
        synchronized(this.z){set(new Object())}
    }
}
```

Ownership types

[OOPSLA'02] Ownership types for safe programming : preventing data races and deadlocks (Boyapati, Lee and Rinard)

Basic idea

Write generic code and create different instance with different protection mechanisms.

Ownership types

Ownership types

$$\text{Each object is owned by } \begin{cases} \text{another object (final field)} \\ \text{itself } (\texttt{self}) \\ \text{a thread } (\texttt{thisThread}) \end{cases}$$

$$\mathit{class}\ \ C\langle \mathit{Owner}_0, \mathit{Owner}_1, \ldots\rangle \qquad \{\ldots \texttt{new}\ D\langle \mathit{Owner}_1\rangle() \ldots\}$$

Ownership types

Static analysis

1. The ownership relation builds a forest of trees
2. The fields of an object must be protected by the ancester of this object (a root, i.e. an object protected by itself or a thread)

Extensions

- support for read-only/single pointer objects

Ownership types

Example

$\texttt{class}\ \mathit{Account}\langle \mathit{thisOwner}\rangle\{$
$\quad \texttt{int}\ \mathit{balance} = 0;$
$\quad \texttt{void}\ \mathit{deposit}(\texttt{int}\ x)\ \texttt{requires}\ \mathit{thisOwner}\{$
$\quad\quad \mathit{this.balance} = \mathit{this.balance} + x;$
$\quad \}$
$\}$
$\mathit{Account}\ \langle \mathit{thisThread}\rangle\ a1 = \texttt{new}\ \mathit{Account}\langle \mathit{thisThread}\rangle$
$a1.\mathit{deposit}(10);$

$\texttt{final}\ \mathit{Account}\langle \mathit{self}\rangle\ a2 = \texttt{new}\ \mathit{Account}\langle \mathit{self}\rangle;$
$\texttt{fork}\{\mathit{synchronized}(a2)\{a2.\mathit{deposit}(10);\}\}$

$\mathit{Account}\langle a_2\rangle\ a3 = \texttt{new}\ \mathit{Account}\langle a_2\rangle;$

Type inference

[VMCAI'04] Type Inference for Parameterized Race-Free Java (Agarwal and Stoller)

idea :

- Perform a set of execution
- Extract types from this set
- Check types

problem : incomplete

[SAS'04, SCP'07] Type inference against races (Flanagan and Freund)

- consider parameterization of classes as introduced by Boyapati,Lee and Rinard
- by reduction of the problem of finding a satisfying assignment for a boolean formula (NP-complete)

Bytecode analysis

[Sigplan Not.] A type system for preventing data races and deadlocks in the java virtual machine language
(Permandla, Roberson, Boyapati)

problem : monitorenter/monitorexit replace synchronized blocks
solution : use indexed types to recover structured locking

Indexed types : [TCS'03] A type system for JVM threads (Laneve)

very simple alias analysis

$$i : \texttt{Load n}, \{\ldots n \mapsto \tau \ldots\}, stack- > \{\ldots n \mapsto \tau_i \ldots\}, \tau_i :: stack$$

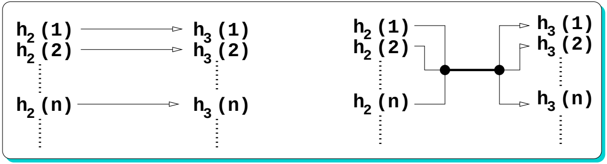
Limitations of type-based approaches

- very strict lock-based discipline
- can't handle other synchronization patterns

Naik and Aiken & al

$$
\begin{array}{l}
a = \texttt{new}\ h_1[N];\\
for(i = 0; i < N; i++)\{\\
\qquad a[i] = new\ \ h_2;\\
\qquad a[i].f = new\ \ h_3;\\
\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{while}\ (*)\{\\
\qquad x = a[*];\\
\qquad \texttt{fork}\{\\
\qquad\qquad \texttt{sync}(x)\{x.f.g = *;\}\\
\qquad \}\\
\}
\end{array}
$$



Language

$$
\begin{array}{rcl}
s & ::= & \mid x = \texttt{null} \mid x = \texttt{new}\ h \\
 & & \mid x = y \mid x = y.f \mid x.f = y \\
 & & \mid s_1; s_2 \mid \texttt{if}\ (*)\ \texttt{then}\ s_1\ \mathit{else}\ s_2 \mid \texttt{while}^w\ (*)\ \texttt{do}\ s
\end{array}
$$

h	allocation site
$w \in \mathbb{W}$	loop counter

Dynamic semantics

$$
\begin{array}{l}
\mathit{Obj} ::= \langle h, \pi \rangle \qquad \pi : \mathbb{W} \to \mathbb{N} \\
C ::= \{ \ldots \mathit{Obj} \triangleright \mathit{Obj}' \ldots \}
\end{array}
$$

Conditional Must Not Aliasing

$$\begin{array}{l} \texttt{synchronized}(x)\{ \\ \quad x.f.g = *; \\ \} \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{synchronized}(y)\{ \\ \quad y.f.g = *; \\ \} \end{array}$$

$$\frac{\text{Conditional Must Not Aliasing}}{must\_not\_alias(x, y) \Rightarrow must\_not\_alias(x.f, y.f)}$$

Disjoint Reachability

Disjoint reachability

$$h \in DR_C(H) \Leftrightarrow \left\{ \begin{array}{l} \overline{o_1}.h \in H \wedge (\overline{o_1} \rhd \overline{o}) \in C^+ \wedge \\ \overline{o_2}.h \in H \wedge (\overline{o_2} \rhd \overline{o}) \in C^+ \wedge \\ \overline{o}.h = h \end{array} \right. \Rightarrow \quad \overline{o_1} = \overline{o_2}$$

Abstraction

$$\begin{array}{lcl} Obj & ::= & \langle \hat{h}, \Pi \rangle \\ \hat{h} & ::= & h \mid \top \\ \Pi & : & \mathbb{W} \mapsto \mathbb{N}_\top \\ \mathbb{N}_\top & = & \{0, 1, \top\} \end{array}$$

$$\begin{array}{ll} \Pi(w) = 0 & w \text{ not active} \\ \Pi(w) = \top & w \text{ unknown} \\ \Pi(w) = \Pi'(w) = 1 & \text{same iteration} \end{array}$$

Judgments : $W, \Pi, \Gamma \vdash s : \Gamma', K$

$h \in DR_K(H)$ $\qquad$ $DR_K(H)$ is a safe appr. of $DR_C(H)$

Disjoint reachability

Examples

$$h_2 \in DR_K(\{h_1\})?$$

$\texttt{while}^1\ (*)\{$
$\quad x = \texttt{new}\ h_1;$
$\quad y = \texttt{new}\ h_2;$
$\quad x.f = y;$
$\}$

$YES : \{\langle h_1, (1)\rangle \triangleright \langle h_2, (1)\rangle\}$

$\texttt{while}^1\ (*)\{$
$\quad x = \texttt{new}\ h_1;$
$\quad \texttt{while}^2\ (*)\{$
$\quad\quad y = \texttt{new}\ h_2;$
$\quad\quad x.f = y;$
$\quad \}$
$\}$

$YES : \{\langle h_1, (1,0)\rangle \triangleright \langle h_2, (1,1)\rangle\}$

$\texttt{while}^1\ (*)\{$
$y = \texttt{new}\ h_2;$
$\quad \texttt{while}^2\ (*)\{$
$\quad x = \texttt{new}\ h_1;$
$\quad x.f = y;$
$\quad \}$
$\}$

$NO : \{\langle h_1, (1,1)\rangle \triangleright \langle h_2, (1,0)\rangle\}$

Disjoint reachability

$$W, \Pi, \Gamma \vdash x = new\ h : \Gamma[x \mapsto \langle \Pi', h \rangle], \emptyset$$

$$W, \Pi, \Gamma \vdash x = y : \Gamma[x \mapsto \Gamma(y)] \qquad W, \Pi, \Gamma \vdash x.f = y, \Gamma, K$$

$$W, \Pi, \Gamma \vdash x = y.f : \Gamma[x \mapsto \langle \lambda w.\top, \top \rangle], \emptyset$$

$$\frac{W, \Pi, \Gamma \vdash s_1 : \Gamma', K_1 \qquad W, \Pi, \Gamma' \vdash s_1 : \Gamma'', K_2}{W, \Pi, \Gamma \vdash s_1; s_2 : \Gamma'', K_1 \cup K_2}$$

$$\frac{W, \Pi, \Gamma \vdash s_1 : \Gamma_1, K_1 \qquad W, \Pi, \Gamma \vdash s_1 : \Gamma_2, K_2}{W, \Pi, \Gamma \vdash \texttt{if}\ (*)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Gamma_1 \sqcup \Gamma_2, K_1 \cup K_2}$$

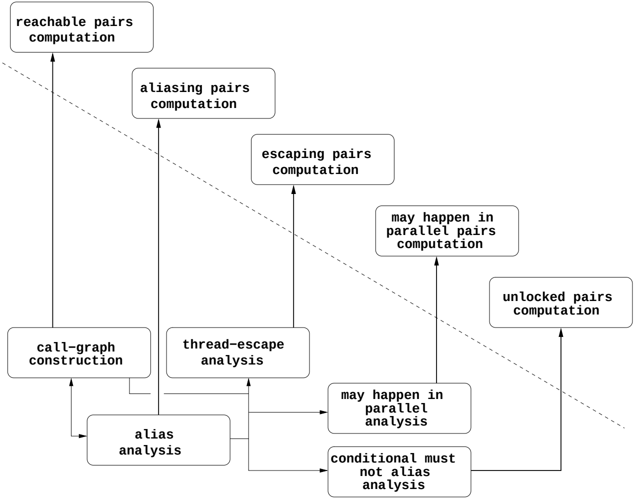
$$\frac{W \cup \{w\}, \Pi, \Gamma^{w^+} \vdash s : \Gamma, K \qquad \Pi(w) \neq 0}{W, \Pi, \Gamma \vdash while^w\ (*)\ \texttt{do}\ s : \Gamma, K}$$

Conflicting pairs elimination

Static analysis

- Call graph construction and context-sensitive points-to analysis
- Type and effect system

Conflicting pairs elimination



Soundness

- reflection
- dynamic loading
- native methods
- libraries ? ? ?

Conclusion and ongoing work

- Data races detection is important
- Objects offer a good framework
- Type systems can handle strict lock-based discipline but lack more elaborated synchronisation patterns
- Points-to analysis can give very precise results but is much more complex

Conclusion and ongoing work

Ongoing work
Certification of a static analysis for data race detection in Coq

1. Context-sensitive points-to analysis
   - Certification of a result checker in Coq

2. Static analysis for data race detection
   - Formalisation of aiken's type and effect system for Java Bytecode
   - Formalisation of successive stages
   - Certification in Coq