Static analysis via abstract interpretation of multithreaded programs

Pietro Ferrara
École Polytechnique
Paris, France
Università Ca' Foscari
Venice, Italy

May 22, 2009

PhD Defense, École Normale Supérieure, Paris, France
Multicore revolution

• The only way to try to prolong Moore's law
• Today: at least dual core processors
• Current trend: manycore
  > Quad cores: 150 € (AMD Phenom X4 9650)
  > Eight cores: server processors (e.g. AMD Opteron)
  > Sixteen cores: soon...
• Sequential programs do not exploit multicores
• Applications with explicit parallelism
"(...) in order for an application to take advantage of the dual-core capabilities, the application should be optimized for multithreading."


- Parallelism supported through **multithreading**
  - Java
  - C#
- Implicit communications via shared memory
- Synchronization on monitors
- **Subtle** and problematic
Motivating Examples

<table>
<thead>
<tr>
<th>ThreadIncrease</th>
<th>Main Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i++;</td>
<td>a.i=0;</td>
</tr>
<tr>
<td></td>
<td>for(int j=0; j&lt;N; j++)</td>
</tr>
<tr>
<td></td>
<td>new ThreadIncrease(a).start();</td>
</tr>
<tr>
<td></td>
<td>if(a.i&gt;1000)</td>
</tr>
<tr>
<td></td>
<td>throw new Exception();</td>
</tr>
</tbody>
</table>

- In order to expose the exception, we need that
  \[ N > 1000 \]
  > Main thread reads a.i after at least 1,000 threads read and increased it

- Really particular execution
  > Exception rarely exposed by testing
  > Difficult to reproduce this execution
### Motivating Examples

<table>
<thead>
<tr>
<th>Deposit 1</th>
<th>Deposit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int t1=a.amount;</td>
<td>int t2=a.amount;</td>
</tr>
<tr>
<td>t1=t1+1000$;</td>
<td>t2=t2+1000$;</td>
</tr>
<tr>
<td>a.amount=t1;</td>
<td>a.amount=t2;</td>
</tr>
</tbody>
</table>

- **Object** a shared between both threads
- **Field** amount declared as volatile
  - All accesses are synchronized
- **No data race**
  - Writes and reads are synchronized
- **Nondeterministic behavior**
  - 1.000$ may “disappear”
  - Particular interleaving of threads' executions
Motivating Examples

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0)</td>
</tr>
<tr>
<td>a.j=1;</td>
<td>throw new Exception();</td>
</tr>
</tbody>
</table>

• At the beginning:
  > i=0
  > j=0

• The exception may be **thrown**
  > Thread 2 may see the values written by Thread 1 in a different order

• Memory model
  > Specify which behaviors are allowed
“Parallel programming is going to require better programming tools to systematically find defects, help debug programs, find performance bottlenecks, and aid in testing. (...) These tools use static program analysis”


- Testing can expose only few multithreaded executions
  - Some executions exposed only by specific VM
  - Difficult to reproduce an execution
- Not sufficient to effectively debug multithreading
- Thus static analysis is appealing for multithreading
  - Infer and prove properties at compile time satisfied by all possible executions

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
Related work

• Much work on analysis of multithreaded programs:
  > Specific properties
    • E.g. data races and deadlocks
  > Not sound for all the possible executions
    • E.g. bounded model checking

• Generic analyzers based on abstract interpretation
  > Successfully applied to sequential programs
  > They can be equipped with several numerical domains
    • Tradeoff between precision and efficiency
  > The same analyzer is instantiated to various properties
  > Sound w.r.t. all the possible executions
Contribution

- Generic approach to the analysis of multithreading
  - Abstract semantics sound w.r.t. a memory model
- Formalization of a specific property
  - Non-determinism due to threads' interleaving
- Framework applied to Java bytecode programs
- Complete implementation
  - Checkmate
    - Generic static analyzer of Java multithreading

- Extension of an existing industrial generic analyzer
  - Specific relational domain to analyze buffer overrun
  - Only for single thread executions
    - Effort in order to apply generic analyzers in practice
Outline

1. Introduction

2. Happens-before memory model
   - Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs
   - Formalization of a specific property

4. Domain and semantics of Java bytecode
   - Low-level domain, specific alias analysis

5. Checkmate
   - Generic sound analyzer of multithreaded programs

6. Static analysis of unsafe code
   - An industrial application of generic analyzer
Memory Model (MM)

- Define which multithreaded behaviors are allowed
  - Restrict non-determinism
  - Allow the most part of
    - compiler optimizations
    - existing virtual machines
    - existing processors

- **Java MM** introduced in 2005, runtime information
- **Happens-before MM** (HBMM) – L. Lamport 1978

- Main components:
  - Program order (intra-thread order of statements)
  - Synchronizes-with relation
HB order and consistency rule

• Happens-before order $\leadsto$:
  > Transitive closure of
  • Program order
  • Synchronizes-with relation

• Core: consistency rule
  > Specify which values written in parallel are visible

• Happens-before consistency rule:
  > A read $r$ of a variable $v$ may see a write $w$ to $v$ if:
    • $\neg(r \leadsto w)$
    • There is no $w'$ to $v$ such that $w \leadsto w' \leadsto r$

• We focus on
  > Mutual exclusion
  > Launch of a thread
Concrete domain and semantics

- **Generic w.r.t. programming language**
- Collect for each thread its trace of execution
  \[ \Psi : Tld \rightarrow St^T \]
- **Abstract away the inter-thread order of execution**
  > Consider each thread separately
- The semantics computes all possible executions
  \[ \langle \mathcal{F}(\Psi), \subseteq, \emptyset, \Psi, \cup, \cap \rangle \]
- **step** function returns the values visible w.r.t. HBMM
- **Intra-thread semantics**
  > Partial trace semantics
  \[ S^o : \Psi \times \Omega \times Tld \rightarrow \mathcal{F}(St^T) \]
  \[ S^o \llbracket f, r, t \rrbracket = lfp^c_0 \lambda T. \{ \sigma_0 \} \cup \{ \sigma_0 \rightarrow \cdots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \sigma_0 \rightarrow \cdots \rightarrow \sigma_{i-1} \in T \land \sigma_i \in \text{step}(t, f, r) \} \]
Concrete semantics

- **Multithread semantics**

\[
\mathcal{S}^\parallel : \Psi \times \Omega \mapsto \wp(\Psi \times \Omega)
\]

\[
\mathcal{S}^\parallel [[f_0, r_0]] = \operatorname{lfp}_\Phi \Phi. \{(f_0, r_0)\} \cup \{(f_i, r) : \exists (f_{i-1}, r) \in \Phi : \\
\forall t \in \operatorname{dom}(f_{i-1}) : f_i(t) \in \mathcal{S}^\parallel [[f_{i-1}, r, t]], \\
f_i(t) = \sigma_0 \rightarrow \cdots \rightarrow \sigma_i, \sigma_i \in \mathcal{S}^\parallel\}
\]

- **Two nested fixpoints**

> Each iteration of the multithread semantics:
  - Produces **new multithreaded executions**
    - That may expose **new values** on shared memory
    - That may produce **new executions** of other threads
    - ...

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
The Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0)</td>
</tr>
<tr>
<td>a.j=1;</td>
<td>throw new Exception();</td>
</tr>
</tbody>
</table>

- 1st iteration of the multithread semantics
  > visible:

  Thread 1:
  
  - i=0
  - j=0
  
  Thread 2:
  
  - i=0
  - j=0
  
  - i=1
  - j=0
  
  - i=1
  - j=1
  
  j==1 && i==0 ? false
The Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0)</td>
</tr>
<tr>
<td>a.j=1;</td>
<td>throw new Exception();</td>
</tr>
</tbody>
</table>

- 2\textsuperscript{nd} iteration of the multithread semantics

> \textit{visible}:

\begin{tabular}{ccccc}
  i=0 & i=1 & i=0 & i=1 \\
  j=0 & j=0 & j=1 & j=1 \\
\end{tabular}

Thread 1:

\begin{tabular}{ccc}
  i=0 & i=1 & i=1 \\
  j=0 & j=0 & j=1 \\
\end{tabular}

Thread 2:

\begin{tabular}{cc}
  i=1 & j=1 && i=0 ? \\
  j=0 & false \\
\end{tabular}

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
The Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0)</td>
</tr>
<tr>
<td>a.j=1;</td>
<td>throw new Exception();</td>
</tr>
</tbody>
</table>

- 2\textsuperscript{nd} iteration of the multithread semantics

> visible: \hspace{1cm}

<table>
<thead>
<tr>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
</tr>
<tr>
<td>j=0</td>
</tr>
<tr>
<td>i=1</td>
</tr>
<tr>
<td>j=0</td>
</tr>
<tr>
<td>i=0</td>
</tr>
<tr>
<td>j=1</td>
</tr>
<tr>
<td>i=1</td>
</tr>
<tr>
<td>j=1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
</tr>
<tr>
<td>j=1</td>
</tr>
<tr>
<td>j==1 &amp;&amp; i==0 ? true</td>
</tr>
</tbody>
</table>

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
## The Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0)</td>
</tr>
<tr>
<td>a.j=1;</td>
<td>throw new Exception();</td>
</tr>
</tbody>
</table>

- **2nd iteration of the multithread semantics**

  > visible:
  
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>i=1</td>
</tr>
<tr>
<td>j=0</td>
<td>j=0</td>
</tr>
<tr>
<td>i=1</td>
<td>i=0</td>
</tr>
<tr>
<td>j=1</td>
<td>j=1</td>
</tr>
</tbody>
</table>

**An execution throws the exception**

- Thread 1: \(j=1\) — \(i=0\) — \(i=1\)
- Thread 2: \(j=1\) — \(i=0\) ?

---

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
Abstract domain and semantics

- Pointwise abstraction of the concrete definitions:
  \[ \overline{\Psi} : \text{TId} \rightarrow \overline{\text{St}}^+ \]

- One trace abstracts all the executions
  > Upper bound of all the visible values
  \[
  \overline{S^\circ} : [(\overline{\Psi} \times \overline{\Omega} \times \text{TId}) \rightarrow \overline{\text{St}}^+] \\
  \overline{S^\circ}[\tilde{f}, \tilde{r}, t] = \text{lfp}_\varepsilon \lambda \tilde{\sigma}. \{\overline{\sigma}_0\} \sqcup_\tau \{\overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} \rightarrow \overline{\sigma}_i : \\
  \overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} = \overline{\tau} \land \overline{\sigma}_i = \text{step}(t, \tilde{f}, \tilde{r}, \overline{\sigma}_{i-1}) \}
  \]

- Sound:
  \[ \forall (\tilde{f}, \tilde{r}) \in \overline{\Psi}_{\text{pre}} \times \overline{\Omega}_{\text{pre}} : \alpha_f(\overline{S}^\parallel)[\tilde{f}, \tilde{r}] \sqsubseteq_f \overline{S}^\parallel[\tilde{f}, \tilde{r}] \]
The Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0)</td>
</tr>
<tr>
<td>a.j=1;</td>
<td>throw new Exception();</td>
</tr>
</tbody>
</table>

• 1st iteration of the multithread semantics

> visible: \[
\begin{align*}
i &= [0..0] \\
j &= [0..0]
\end{align*}
\]

Thread 1: \[
\begin{align*}
i &= [0..0] & i &= [1..1] & i &= [1..1] \\
j &= [0..0] & j &= [0..0] & j &= [1..1]
\end{align*}
\]

Thread 2: \[
\begin{align*}
i &= [0..0] & j &= 1 \land i &= 0 \ ? \\
j &= [0..0] & \quad & \quad & false
\end{align*}
\]
The Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1; a.j=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0) throw new Exception();</td>
</tr>
</tbody>
</table>

- 2nd iteration of the multithread semantics

> visible:

\[
\begin{align*}
  i & = [0..1] \\
  j & = [0..1]
\end{align*}
\]

Thread 1:

\[
\begin{align*}
  i & = [0..0] \\
  j & = [0..0] \\
  i & = [1..1] \\
  j & = [0..0] \\
  i & = [1..1] \\
  j & = [1..1]
\end{align*}
\]

Thread 2:

\[
\begin{align*}
  i & = [0..1] \\
  j & = [0..1] \\
  j & = 1 \\
  & \text{&& } i = 0 \ ? \\
  & \text{top}
\end{align*}
\]
The Example

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.i=1; a.j=1;</td>
<td>if(a.j==1 &amp;&amp; a.i==0) throw new Exception();</td>
</tr>
</tbody>
</table>

- **2nd iteration of the multithread semantics**
  > visible: 
  
  \[
  \begin{align*}
  i &= [0..1] \\
  j &= [0..1]
  \end{align*}
  \]

Thread 1:
\[
\begin{align*}
  i &= [0..0] \\
  j &= [0..0] \\
  i &= [1..1] \\
  j &= [0..0] \\
  i &= [1..1] \\
  j &= [1..1] \\
  i &= [0..1] \\
  j &= [0..1]
  \end{align*}
\]

Thread 2:
\[
\begin{align*}
  j &= 1 \land i = 0 \land \\
  &\text{top} \\
  i &= [0..1] \\
  j &= [0..1]
  \end{align*}
\]

The program may throw the exception
Outline

1. Introduction

2. Happens-before memory model
   - Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs
   - Formalization of a specific property

4. Domain and semantics of Java bytecode
   - Low-level domain, specific alias analysis

5. Checkmate
   - Generic sound analyzer of multithreaded programs

6. Static analysis of unsafe code
   - An industrial application of generic analyzer
An example

<table>
<thead>
<tr>
<th>Deposit1</th>
<th>Deposit2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int t1=a.amount;</td>
<td>int t2=a.amount;</td>
</tr>
<tr>
<td>t1=t1+1000$;</td>
<td>t2=t2+1000$;</td>
</tr>
<tr>
<td>a.amount=t1;</td>
<td>a.amount=t2;</td>
</tr>
</tbody>
</table>

- At the end, do we deposit 1.000$ or 2.000$?  
- It depends on threads' interleaving:  
  > If Deposit1 and Deposit2 read the initial amount:  
    • 1.000$  
  > If we execute e.g. first Deposit1 and then Deposit2:  
    • 2.000$  
- Different values because of arbitrary interleaving
Our solution

- **Statically** analyze the determinism
  - Focused on communications on shared memory
  - **Generic** w.r.t.
    - Programming language
    - Numerical domain
    - Memory model

- **Advantages**
  - Deal *directly* with the effects of arbitrary interleaving
  - Flexible
Concrete domain and property

\[ S : [\text{Var} \rightarrow (V \times \text{TId})] \]

- Each value is related to a thread identifier
  - Trace which thread wrote it in the shared memory

- A program is not deterministic iff
  - two executions
    - of the same thread
    - in the same position of the traces of execution
  - contain two shared memories
    - in which the same variable contains values related to different thread identifiers

\[ ds(s_1, s_2) = \text{false} \]
\[ \uparrow \]
\[ \exists \text{var} \in \text{dom}(s_1) \cap \text{dom}(s_2) : s_1(\text{var}) = (\text{val}_1, t_1), s_2(\text{var}) = (\text{val}_2, t_2), t_1 \neq t_2 \]
An example

**Thread Deposit1:**

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>10.000$</td>
<td>System</td>
</tr>
</tbody>
</table>

**Thread Deposit2:**

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>11.000$</td>
<td>Deposit1</td>
</tr>
</tbody>
</table>

---

**Thread Deposit1:**

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>10.000$</td>
<td>System</td>
</tr>
</tbody>
</table>

**Thread Deposit2:**

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>11.000$</td>
<td>Deposit1</td>
</tr>
</tbody>
</table>

---

**Thread Deposit1:**

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>10.000$</td>
<td>System</td>
</tr>
</tbody>
</table>

**Thread Deposit2:**

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>11.000$</td>
<td>Deposit2</td>
</tr>
</tbody>
</table>
### An example

#### Thread Deposit1:

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>10.000$</td>
<td>System</td>
</tr>
</tbody>
</table>

#### Thread Deposit2:

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>11.000$</td>
<td>Deposit1</td>
</tr>
</tbody>
</table>

#### Thread Deposit1:

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>10.000$</td>
<td>System</td>
</tr>
</tbody>
</table>

#### Thread Deposit2:

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Value</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>11.000$</td>
<td>Deposit1</td>
</tr>
</tbody>
</table>

### Non-deterministic executions!
Levels of abstraction

- **First level**: $\hat{S} : [\text{Var} \rightarrow [\text{TId} \rightarrow \hat{V}]]$
  - Parameterized by an abstract numerical domain $\hat{V}$
    > One value for each thread

- **Second level**: $\bar{S} : [\text{Var} \rightarrow (\hat{V} \times \phi(\text{TId}))]$
  - Trace
    > One abstract value
    > The set of threads that may have written it

- **Sound**

\[
\langle \phi(\Psi), \subseteq \rangle \xleftrightarrow[\alpha_{\Psi}]{\gamma_{\Psi}} \langle \hat{\Psi}, \subseteq_{\hat{\Psi}} \rangle \xleftrightarrow[\alpha_{\hat{\Psi}}]{\gamma_{\hat{\Psi}}} \langle \overline{\Psi}, \subseteq_{\overline{\Psi}} \rangle
\]
Determinism on abstract states

- First abstraction
  \[ \widehat{ds}(\widehat{s}) = \text{false} \]
  \[ \Leftrightarrow \exists \text{var} \in \text{dom}(\widehat{s}) : |\text{dom}(\widehat{s} (\text{var} ))| > 1 \]

- Second abstraction
  \[ \overline{ds}(\overline{s}) = \text{false} \iff \exists \text{var} \in \text{dom}(\overline{s}) : |\pi_2(\overline{s}(\text{var} ))| > 1 \]

- Soundness
  \[ \forall \theta \in \wp(\Psi) : d(\theta) = \text{false} \Rightarrow \widehat{d}(\alpha_\Psi(\theta)) = \text{false} \]
  \[ \forall f \in \Psi : \widehat{d}(f) = \text{false} \Rightarrow \overline{d}(\alpha_{\overline{\Psi}}(f)) = \text{false} \]
### An example – 1\textsuperscript{st} abstraction

#### Thread Deposit1:

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>System</td>
<td>[10.000..10.000]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>Deposit1</td>
<td>[11.000..11.000]$</td>
</tr>
</tbody>
</table>

#### Thread Deposit2:

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>System</td>
<td>[10.000..10.000]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>Deposit1</td>
<td>[11.000..11.000]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obj.</th>
<th>Field</th>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>Deposit2</td>
<td>[11.000..12.000]$</td>
</tr>
</tbody>
</table>
### An example – 1st abstraction

**Thread Deposit1:**

<table>
<thead>
<tr>
<th>Obj. Field</th>
<th>Field</th>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>System</td>
<td>[10.000..10.000]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deposit1</td>
<td>[11.000..11.000]</td>
</tr>
</tbody>
</table>

**Thread Deposit2:**

<table>
<thead>
<tr>
<th>Obj. Field</th>
<th>Field</th>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>amount</td>
<td>System</td>
<td>[10.000..10.000]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deposit1</td>
<td>[11.000..11.000]</td>
</tr>
</tbody>
</table>

Non-deterministic program!
Weak determinism

\[ \overline{wds} : [\overline{S} \rightarrow \{\text{true, false}\}] \]
\[ \overline{wds} (\overline{S}) = \text{false} \]
\[ \Leftrightarrow \exists \text{var} \in \text{dom}(\overline{S}) : |\text{dom}(\overline{S}(\text{var}))| > 1 \]
\[ \land \exists t_1, t_2 \in \text{dom}(\overline{S}(\text{var})) : \overline{S}(\text{var})(t_1) \neq \overline{S}(\text{var})(t_2) \]

• Relax the full determinism
  > On the first level of abstraction
  > Rely on a numerical abstract domain

• It allows non deterministic behaviors iff
  > The abstract values written in parallel by different threads are the same
    • E.g. if the sign of the values is the same

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
Projecting states and traces

- Check the determinism only
  - On a subset of the shared variables
    - Only the amount of the bank account
  - On a subset of the trace
    - Only the actions that deposit or withdraw money
Outline

1. Introduction

2. Happens-before memory model
   • Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs
   • Formalization of a specific property

4. Domain and semantics of Java bytecode
   • Low-level domain, specific alias analysis

5. Checkmate
   • Generic sound analyzer of multithreaded programs

6. Static analysis of unsafe code
   • An industrial application of generic analyzer

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
From theory to practice

- **Ultimate goal**
  > Develop a static analysis of Java programs

- **Theoretical approach:**
  > Set of thread identifiers
  > Set of shared locations
  > Set of synchronizable elements

- **From theory to... Java!**
  > Threads: objects
  > Shared memory: heap
  > Synchronizable elements: monitors on objects
Features

- We support:
  - Dynamic allocation of shared memory
  - Dynamic creation and launch of threads
  - Dynamic creation of monitors
- In addition, common Java features like:
  - Strings
  - Arrays
  - Static fields and methods
  - Overload, overriding, recursion
- We fully support the Java bytecode language
Concrete domain

- **Low-level domain**
  - Simulate the Java Virtual Machine (JVM)
  - Based on the JVM specification
    - Operand stack: \( \text{Op} = \mathcal{S} T(\text{Val}) \)
    - Heap: \( \mathcal{H} : [\text{Ref} \rightarrow (\text{Obj} \cup \text{Arr} \cup \text{Str})] \)
    - Local variables: \( \mathcal{L V} = \mathcal{A R}(\text{Val}) \)
    - Locked monitors: \( \mathcal{L} : [\text{Ref} \rightarrow \mathbb{N}] \)

- We represent programs as **Control Flow Graph** (CFG)

- 1\(^{st}\) abstraction:
  - Executions on the Control Flow Graph (exCFG)
  - **Sound** abstraction of real executions

\[
\langle \phi(T), \subseteq \rangle \xleftarrow{\gamma_{\text{CFG}}} \alpha_{\text{CFG}} \xrightarrow{\gamma_{\text{CFG}}} \langle \text{exCFG}, \subseteq_{\text{CFG}} \rangle
\]

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
Alias analysis

- Concrete references: potentially infinite
- We need to check when references
  - May point to the same location (may-aliasing)
  - Always point to the same location (must-aliasing)

- May aliasing:
  - Bound each reference to the program point that allocates it

- Must aliasing:
  - Each reference related to an equivalence class
  - Rough but precise enough
Abstract domain and semantics

• Other components: (almost) pointwise abstraction
  > Operand stack: \( \overline{\text{Op}} = \$T(\overline{\text{Val}}) \)
  > Heap: \( \overline{\text{H}} : [P \to (\text{Obj} \cup \text{Arr} \cup \text{Str})] \)
  > Local variables: \( \overline{\text{LV}} = \overline{\text{AR}}(\overline{\text{Val}}) \)
  > Locked monitors: \( \overline{\text{L}} : [\text{Ref} \to \mathbb{N}] \)

• Proved the soundness
  \[
  \langle \phi(\Sigma), \subseteq \rangle \xrightarrow{\gamma_\Sigma} \overline{\langle \Sigma, \subseteq \rangle}
  \]

• Operational semantics of statements
• Proved the local soundness
  \[
  \forall \sigma \in \Sigma : \alpha_\Sigma(\{\sigma' : \sigma \rightarrow \sigma'\}) \subseteq_\Sigma \overline{\sigma'} : \alpha_\Sigma(\{\sigma\}) \rightarrow \overline{\sigma'}
  \]

• Applied to HBMM and determinism
1. Introduction

2. Happens-before memory model
   • Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs
   • Formalization of a specific property

4. Domain and semantics of Java bytecode
   • Low-level domain, specific alias analysis

5. Checkmate
   • Generic sound analyzer of multithreaded programs

6. Static analysis of unsafe code
   • An industrial application of generic analyzer
Checkmate

- **Generic analyzer of multithreaded program**
  - Sound
  - Flow-sensitive
- **Generic w.r.t.**
  - Numerical domain
    - Interval, sign, parity, congruence
  - Memory model
    - Happens-before memory model
- **Property of interest**
  - Multithreading: data race, deadlock, determinism
  - Well-known: division by zero, access to null, etc..

http://www.pietro.ferrara.name/checkmate
A data race may happen at line 5 of class Temp when executed by thread th1. The value may be written in parallel by thread th2 when executing line 16 of class MyThread.
Experimental results

• Applied to
  > A set of examples taken from [JMM]
  > Some case studies taken from [LEA]
  > A family of applications of increasing size
  > Some benchmarks taken from [PRA, BENCH]

• Fast for small programs

• Precise
  > But not scalable for large/industrial programs

## External benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>St.</th>
<th>Th.</th>
<th>Top</th>
<th>Sign</th>
<th>Int.</th>
<th>Par.</th>
<th>Cong.</th>
</tr>
</thead>
<tbody>
<tr>
<td>philo</td>
<td>213</td>
<td>2</td>
<td>&lt;1”</td>
<td>&lt;1”</td>
<td>1”</td>
<td>&lt;1”</td>
<td>&lt;1”</td>
</tr>
<tr>
<td>forkjoin</td>
<td>170</td>
<td>2</td>
<td>&lt;1”</td>
<td>&lt;1”</td>
<td>&lt;1”</td>
<td>&lt;1”</td>
<td>&lt;1”</td>
</tr>
<tr>
<td>barrier</td>
<td>363</td>
<td>3</td>
<td>&lt;1”</td>
<td>1”</td>
<td>2”</td>
<td>1”</td>
<td>1”</td>
</tr>
<tr>
<td>sync</td>
<td>320</td>
<td>3</td>
<td>1”</td>
<td>1”</td>
<td>3”</td>
<td>1”</td>
<td>2”</td>
</tr>
<tr>
<td>crypt</td>
<td>2636</td>
<td>3</td>
<td>5”</td>
<td>6”</td>
<td>17”</td>
<td>6”</td>
<td>5”</td>
</tr>
<tr>
<td>sor</td>
<td>1121</td>
<td>2</td>
<td>4”</td>
<td>7”</td>
<td>17”</td>
<td>6”</td>
<td>5”</td>
</tr>
<tr>
<td>elevator</td>
<td>1829</td>
<td>2</td>
<td>31”</td>
<td>11”</td>
<td>19”</td>
<td>30”</td>
<td>29”</td>
</tr>
<tr>
<td>lufact</td>
<td>3732</td>
<td>2</td>
<td>27”</td>
<td>53”</td>
<td>5’59”</td>
<td>29”</td>
<td>29”</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3864</td>
<td>2</td>
<td>1’02”</td>
<td>2’35”</td>
<td>1h00’56”</td>
<td>1’43”</td>
<td>1’04”</td>
</tr>
</tbody>
</table>
Outline

1. Introduction

2. Happens-before memory model
   - Definition in fixpoint form and abstraction

3. Determinism of multithreaded programs
   - Formalization of a specific property

4. Domain and semantics of Java bytecode
   - Low-level domain, specific alias analysis

5. Checkmate
   - Generic sound analyzer of multithreaded programs

6. Static analysis of unsafe code
   - An industrial application of generic analyzer

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
Generic analyzers

- Extend an industrial generic analyzer
  - Clousot - Microsoft Research
  - Sound only at single-thread level
    - Effort to apply generic analyzers to a property
    - Show practical interest of this type of analyzers
    - Future work:
      - Apply Checkmate to industrial programs
Background

- .Net: safe environment of execution
  - Exception: unsafe code
  - Direct access to the memory
- No guarantees on direct memory accesses
  - Buffer overrun
- Goal: apply Clousot to unsafe code
- Combination with contracts
  - Method boundary annotation
  - Lightweight domains
    - Stripes: new relational domain
    - Combined with well-known numerical domains
  - Scalability
    - .Net libraries analyzed in a couple of minutes
WB(p) ≥ n * (count[+base]) + k

- Relational
- Precise

• **Linear** complexity in practice
Example: InitToZero

\[
\text{check}(\text{WB}(p) \geq \text{sizeof}(x) + \exp \ast \text{sizeof}(*p), \bar{s}) = \text{true} \\
\text{check}(\exp \geq 0, \bar{s}) = \text{true} \\
\text{F}[*(p + \exp) = x](\bar{s}) \rightarrow \bar{s}
\]

unsafe void InitToZero(int* ptr, uint len)
{
    Contract.Requires(Contract.WB(ptr) \geq \text{len}\ast4);
    for (int i = 0; i < \text{len}; i++)
       *(ptr + i) = 0;
}
Example: \texttt{InitToZero}

\begin{align*}
\text{check}(\text{WB}(p) \geq \text{sizeof}(x) + \text{exp} \times \text{sizeof}(\ast p), \overline{s}) &= \text{true} \\
\text{check}(\text{exp} \geq 0, \overline{s}) &= \text{true}
\end{align*}

\[
\mathbf{F}[\forall (p + \text{exp}) = x](\overline{s}) \rightarrow \overline{s}
\]

unsafe void \texttt{InitToZero}(int* ptr, uint len)
{
    Contract.Requires(Contract.WB(ptr) \geq \text{len}\times4);
    for (int i = 0; i < \text{len}; i++)
        *(ptr + i) = 0;
}

Infer:
\begin{align*}
\text{WB}(p) &\geq 4 \times \text{len} \\
\text{len} &\geq i + 1 \\
\text{WB}(p) &\geq 4 \times i + 4
\end{align*}
Example: InitToZero

```
unsafe void InitToZero(int* ptr, uint len)
{
    Contract.Requires(Contract.WB(ptr) ≥ len*4);
    for (int i = 0; i < len; i++)
        *(ptr + i) = 0;
}
```

Proof obligation:

\[ \text{WB(ptr)} \geq 4 + i \times 4 \]

Infer:

\[
\text{WB(ptr)} \geq 4 \times \text{len} \\
\text{len} \geq i + 1 \\
\text{WB(ptr)} \geq 4 \times i + 4
\]
Example: `InitToZero`

```c
unsafe void InitToZero(int* ptr, uint len)
{
    Contract.Requires(Contract.WB(ptr) ≥ len*4);
    for (int i = 0; i < len; i++)
        *(ptr + i) = 0;
}
```

Proof obligation:

\[ \text{WB}(p) \geq \text{sizeof}(x) + \exp \times \text{sizeof}(*p), \overline{s} = \text{true} \]
\[ \text{WB}(\exp \geq 0), \overline{s} = \text{true} \]

\[ \overline{F}[*(p + \exp) = x](\overline{s}) \rightarrow \overline{s} \]

Validate:

\[ \text{WB}(p) \geq 4 \times \text{len} \]
\[ \text{len} \geq i + 1 \]
\[ \text{WB}(p) \geq 4 \times i + 4 \]
Intervals

\[
\begin{align*}
\text{check}(\text{WB}(p)) & \geq \text{sizeof}(x) + \exp \times \text{sizeof}(\ast p), \overline{s}) = \text{true} \\
\text{check}(\exp \geq 0, \overline{s}) & = \text{true} \\
\overline{F}[\ast(p + \exp) = x](\overline{s}) & \rightarrow \overline{s}
\end{align*}
\]

for (int i=0; i<len; i++)
\[
\ast(p + i) = 0;
\]

• Stripes do not prove \( i \geq 0 \)
check(WB(p) \geq \text{sizeof(x)} + \exp \cdot \text{sizeof(*p)}, \bar{s}) = \text{true}
check(\exp \geq 0, \bar{s}) = \text{true}

\[ F[(*p + \exp) = x](\bar{s}) \rightarrow \bar{s} \]

for (int i=0; i<len; i++)
*(ptr + i) = 0;

- Stripes do not prove \( i \geq 0 \)

for (int i = 0; i < len; i++)
*(ptr + i) = 0;

- Intervals do it!
## Experimental Results

<table>
<thead>
<tr>
<th>Assembly</th>
<th>#Methods</th>
<th>Time</th>
<th>Checked</th>
<th>Val.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>mscorlib.dll</td>
<td>18084</td>
<td>3m43s</td>
<td>3069</td>
<td>1835</td>
<td>59.79%</td>
</tr>
<tr>
<td>System.dll</td>
<td>13776</td>
<td>3m18s</td>
<td>1720</td>
<td>1048</td>
<td>60.93%</td>
</tr>
<tr>
<td>System.Data.dll</td>
<td>11333</td>
<td>3m45s</td>
<td>138</td>
<td>59</td>
<td>42.75%</td>
</tr>
<tr>
<td>System.Design.dll</td>
<td>11419</td>
<td>2m42s</td>
<td>16</td>
<td>10</td>
<td>62.50%</td>
</tr>
<tr>
<td>System.Drawing.dll</td>
<td>3120</td>
<td>19s</td>
<td>48</td>
<td>29</td>
<td>60.42%</td>
</tr>
<tr>
<td>System.Web.dll</td>
<td>22076</td>
<td>3m19s</td>
<td>88</td>
<td>44</td>
<td>50.00%</td>
</tr>
<tr>
<td>System.Windows.Forms.dll</td>
<td>23180</td>
<td>4m31s</td>
<td>364</td>
<td>266</td>
<td>73.08%</td>
</tr>
<tr>
<td>System.XML.dll</td>
<td>10046</td>
<td>2m41s</td>
<td>772</td>
<td>311</td>
<td>40.28%</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57.96%</td>
</tr>
</tbody>
</table>

- **Scalable** analysis
- Code not annotated, **false alarms**
- System.Drawing exposes warnings on 5 methods
  - **Bug**
    - Public method
    - It causes the crash at runtime

Pietro Ferrara: “Static analysis via abstract interpretation of multithreaded programs”
PhD Defense, École Normale Supérieure, Paris, France
Conclusion

• **Generic static analysis of multithreaded programs**
  > Based on abstract interpretation
  > Applied to a real programming language
    • **Bytecode** level, it can analyze other languages
      > E.g. Scala
  > Implementation
    • Experimental results encouraging
    • **Scalability** is still an open issue

• Other generic analyzers scale up
  > **Local reasoning**, i.e. not whole program analyses
  > Based on method boundary annotations
Future work

• How to refine the MM and its analysis
  > Other synchronizations have to be considered
  > Interesting restrictions of the Java MM
• How to relax the property of determinism
• Refine bytecode domain and semantics
  > Goal: apply to numerical relational domains
• Implement it in Checkmate
Future work

- Whole program analysis
  - Limit: it does not scale!
- Modular reasoning
  - Impossible on multithreaded programs
  - Lack of programming languages and contracts
- Object-oriented programs
  - Restrict the visibility of fields and methods
    - public, private, protected
  - Contracts on classes and methods
- Intuition
  - Apply and tune these ideas to multithreading
Publications


Thank you!