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

Multithreading

> *"(…) in order for an application to take advantage of the dual-core capabilities, the application should be optimized for multithreading."*

G. Koch. *Discovering multi-core: extending the benefits of Moore's law.* In *Technology Intel Magazine*. Intel, July 2005.

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

Motivating Examples

ThreadIncrease

```
a.i++;
```

Main Thread

```
a.i=0;
for(int j=0; j<N; j++)
  new ThreadIncrease(a).start();
if(a.i>1000)
  throw new Exception();
```

- 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

Deposit 1	**Deposit 2**

`int t1=a.amount;`
`t1=t1+1000$;`
`a.amount=t1;`
	`int t2=a.amount;`
	`t2=t2+1000$;`
	`a.amount=t2;`

- 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

Thread 1	Thread 2
`a.i=1;` `a.j=1;`	`if(a.j==1 && a.i==0)` `throw new Exception();`

- 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

Static analysis

> *"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"*

Herb Sutter and James Larus. *Software and the concurrency revolution*. In *ACM Queue. ACM Press, 2005*.

- 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

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 $\rightsquigarrow$:
  - 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:
    - $\mathtt{NOT}(r \rightsquigarrow w)$
    - There is no w' to v such that $w \rightsquigarrow w' \rightsquigarrow r$
- 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

$$\Psi : \mathsf{TId} \rightarrow \mathsf{St}^{\vec{+}}$$

- Abstract away the inter-thread order of execution
  - Consider each thread separately
- The semantics computes all possible executions

$$\langle \wp(\Psi), \subseteq, \emptyset, \Psi, \cup, \cap \rangle$$

- *step* function returns the values visible w.r.t. HBMM
- Intra-thread semantics
  - Partial trace semantics

$$\mathbb{S}^{\circ} : \Psi \times \Omega \times \mathsf{TId} \mapsto \wp(\mathsf{St}^{\vec{+}})$$

$$\mathbb{S}^{\circ}[\![\mathsf{f}, \mathsf{r}, \mathsf{t}]\!] = \mathit{lfp}_{\emptyset}^{\subseteq} \lambda \mathsf{T}.\{\sigma_0\} \cup \{\sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{i}-1} \rightarrow \sigma_{\mathsf{i}} : \sigma_0 \rightarrow \cdots \rightarrow \sigma_{\mathsf{i}-1} \in \mathsf{T} \wedge \sigma_{\mathsf{i}} \in \mathit{step}(\mathsf{t}, \mathsf{f}, \mathsf{r})\}$$

Concrete semantics

- Multithread semantics

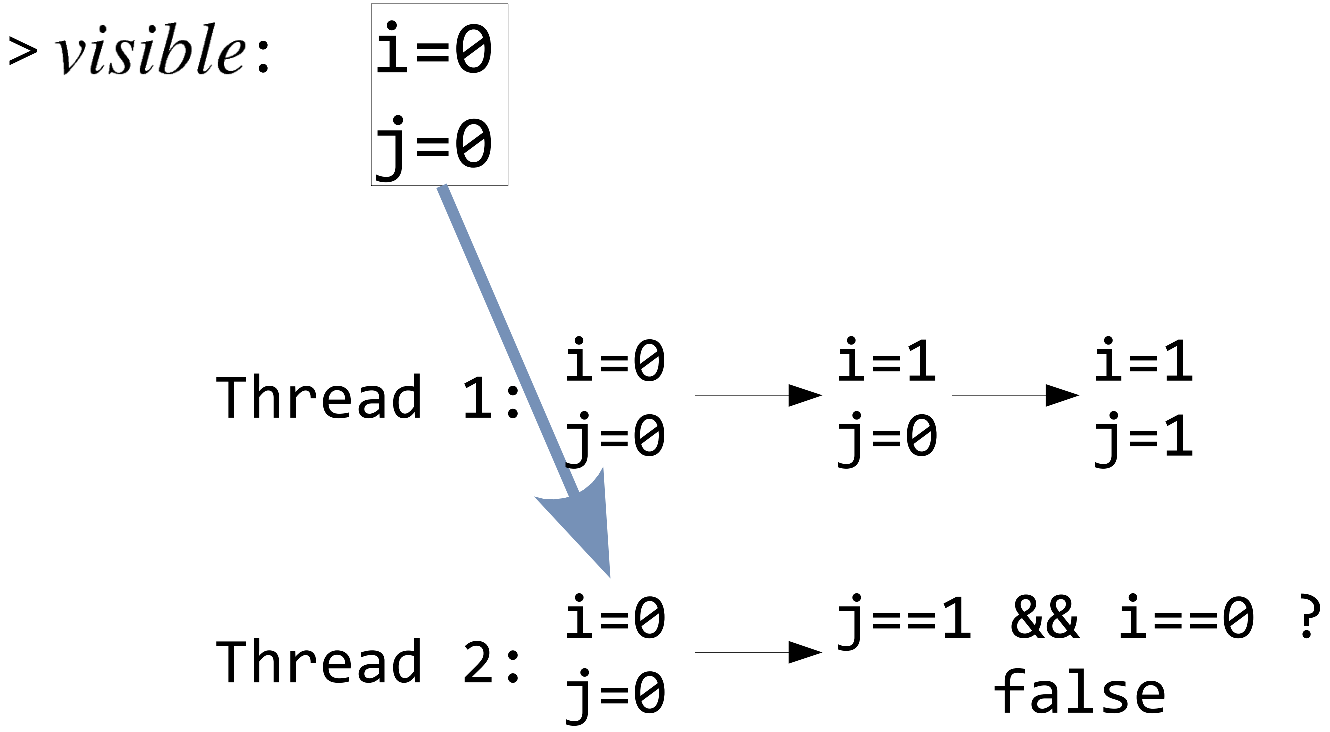
$$\begin{aligned}
&\mathbb{S}^{\|} : \Psi \times \Omega \mapsto \wp(\Psi \times \Omega)\\
&\mathbb{S}^{\|}[\![f_0, r_0]\!] = \mathit{lfp}_{\emptyset}^{\subseteq} \lambda\Phi.\{(f_0, r_0)\} \cup \{(f_i, r) : \exists (f_{i-1}, r) \in \Phi :\\
&\qquad\qquad \forall t \in \mathit{dom}(f_{i-1}) : f_i(t) \in \mathbb{S}^{\circ}[\![f_{i-1}, r, t]\!],\\
&\qquad\qquad f_i(t) = \sigma_0 \to \cdots \to \sigma_i, \sigma_i \in St_{\overset{\circ}{\to}}\}
\end{aligned}$$

- 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
      - > …

The Example

Thread 1	Thread 2
`a.i=1;` `a.j=1;`	`if(a.j==1 && a.i==0)` `throw new Exception();`

- 1^{st} iteration of the multithread semantics



> *visible*:

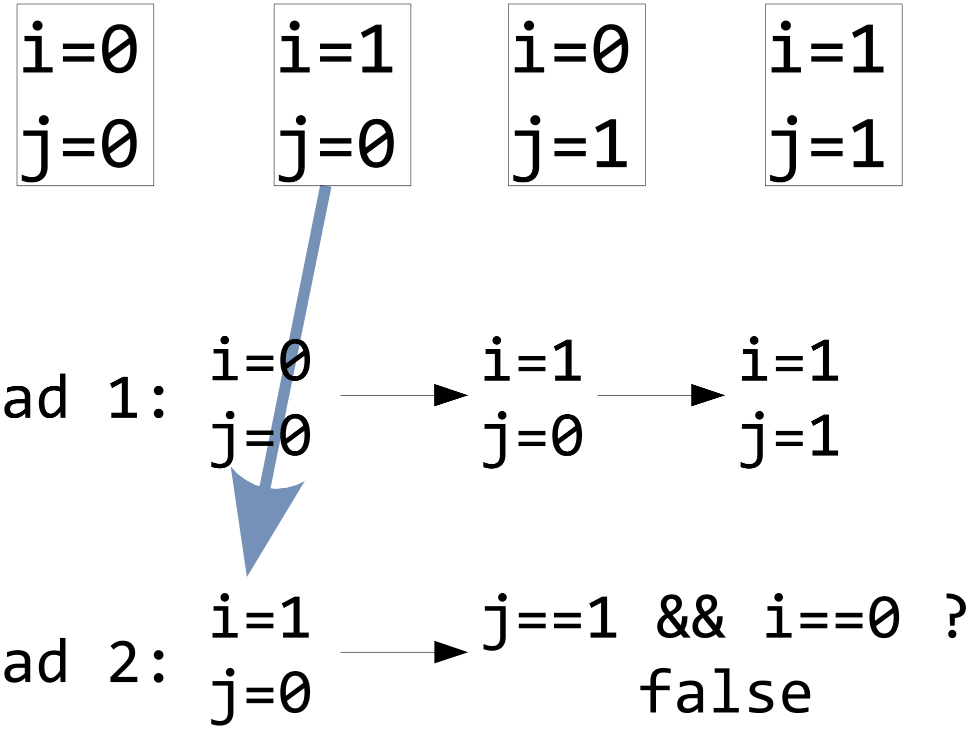
```
i=0
j=0
```

```
Thread 1:  i=0 ──► i=1 ──► i=1
           j=0     j=0     j=1

Thread 2:  i=0 ──► j==1 && i==0 ?
           j=0          false
```

The Example

Thread 1	**Thread 2**
`a.i=1;` `a.j=1;`	`if(a.j==1 && a.i==0)` ` throw new Exception();`

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

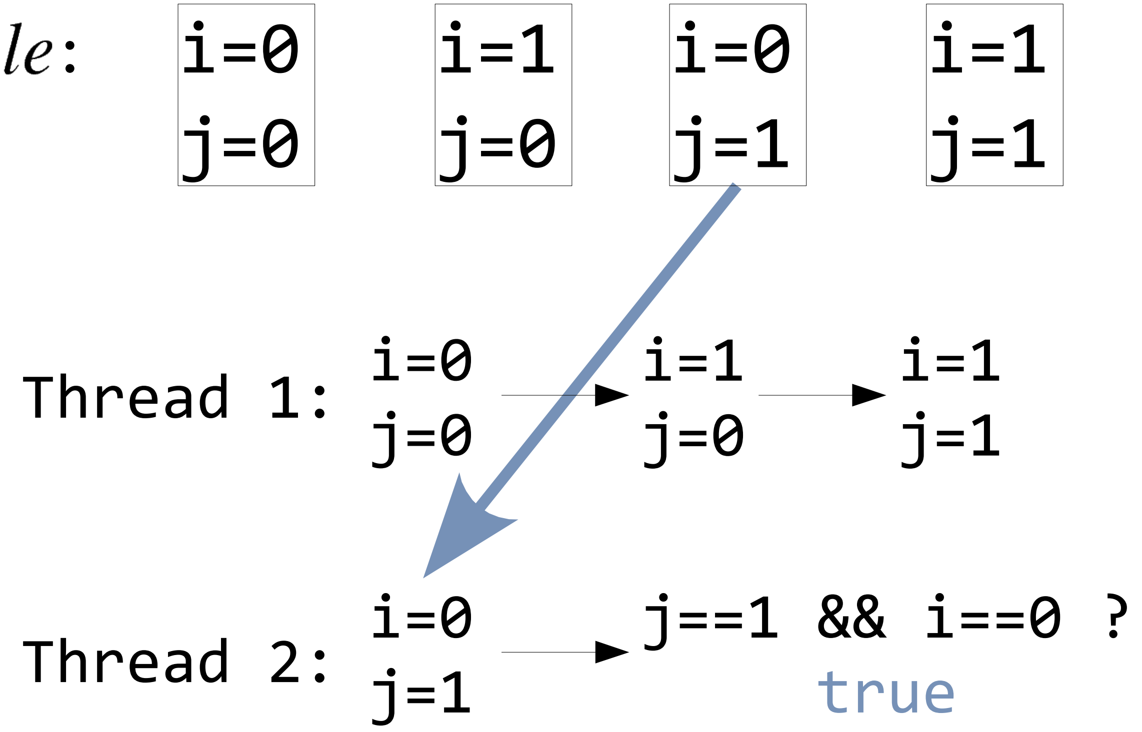


```
i=0    i=1    i=0    i=1
j=0    j=0    j=1    j=1

Thread 1:  i=0  →  i=1  →  i=1
           j=0     j=0     j=1

Thread 2:  i=1  →  j==1 && i==0 ?
           j=0          false
```

The Example

Thread 1	Thread 2
`a.i=1;` `a.j=1;`	`if(a.j==1 && a.i==0)` `throw new Exception();`

- 2nd iteration of the multithread semantics
  - > *visible*:



i=0 j=0

i=1 j=0

i=0 j=1

i=1 j=1

Thread 1: i=0 j=0 → i=1 j=0 → i=1 j=1

Thread 2: i=0 j=1 → j==1 && i==0 ? true

The Example

Thread 1

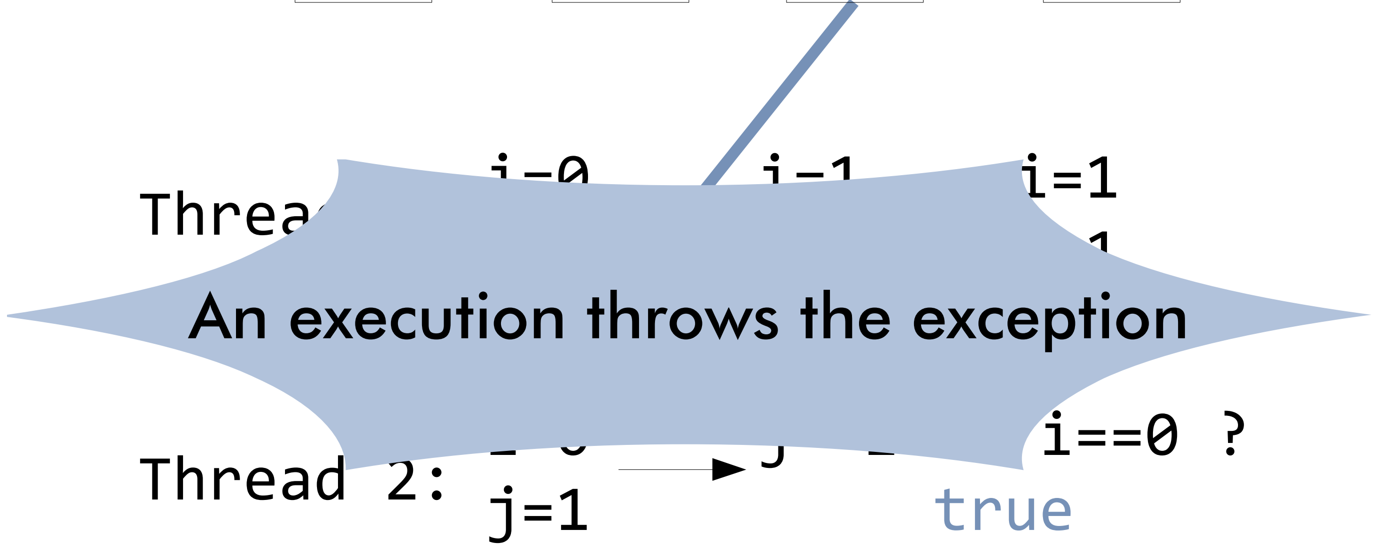
```
a.i=1;
a.j=1;
```

Thread 2

```
if(a.j==1 && a.i==0)
 throw new Exception();
```

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

`i=0` `j=0`   `i=1` `j=0`   `i=0` `j=1`   `i=1` `j=1`



An execution throws the exception

Abstract domain and semantics

- Pointwise abstraction of the concrete definitions:

$$\overline{\Psi} : \mathsf{TId} \rightarrow \overline{\mathsf{St}}^{\vec{+}}$$

- One trace abstracts all the executions
  - > Upper bound of all the visible values

$$\overline{\mathbb{S}^{\circ}} : [(\overline{\Psi} \times \overline{\Omega} \times \mathsf{TId}) \rightarrow \overline{\mathsf{St}}^{\vec{+}}]$$

$$\overline{\mathbb{S}^{\circ}}[\![\bar{\mathsf{f}}, \bar{\mathsf{r}}, \mathsf{t}]\!] = \mathit{lfp}_{\epsilon}^{\sqsubseteq_{\tau}} \lambda\overline{\tau}.\{\overline{\sigma}_0\} \sqcup_{\tau} \{\overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} \rightarrow \overline{\sigma}_i : \\ \overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} = \overline{\tau} \wedge \overline{\sigma}_i = \overline{\mathit{step}}(\mathsf{t}, \bar{\mathsf{f}}, \bar{\mathsf{r}}, \overline{\sigma}_{i-1})\}$$

$$\overline{\mathbb{S}^{\parallel}} : [\overline{\Psi} \times \overline{\Omega} \rightarrow \overline{\Psi} \times \overline{\Omega}]$$

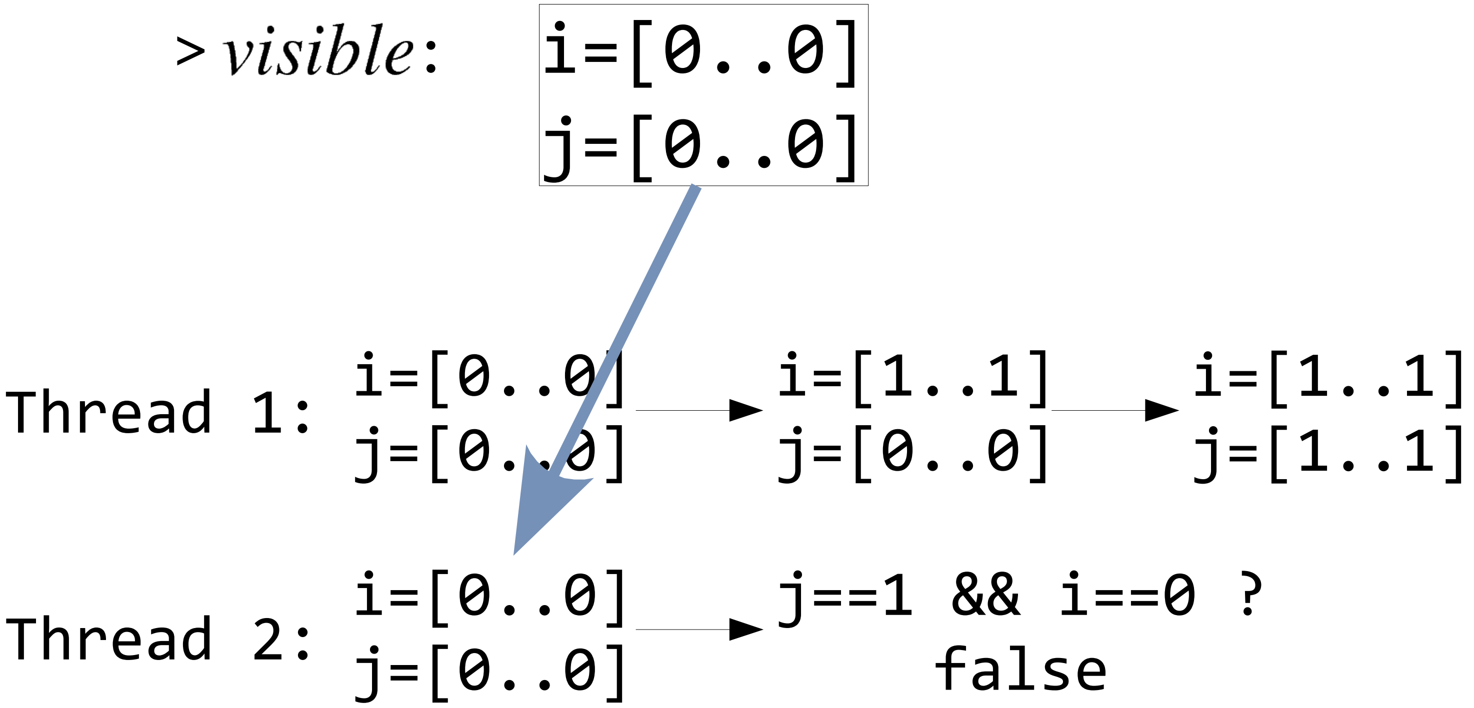
$$\overline{\mathbb{S}^{\parallel}}[\![\bar{\mathsf{f}}_0, \bar{\mathsf{r}}_0]\!] = \mathit{lfp}_{\emptyset}^{\sqsubseteq_{f}} \lambda(\bar{\mathsf{f}}, \bar{\mathsf{r}}).\{(\bar{\mathsf{f}}_0, \bar{\mathsf{r}}_0)\} \sqcup_{f} \{(\bar{\mathsf{f}}_i, \bar{\mathsf{r}}) : \forall \mathsf{t} \in \mathit{dom}(\bar{\mathsf{f}}) : \\ \bar{\mathsf{f}}_i(\mathsf{t}) = \overline{\mathbb{S}^{\circ}}[\![\bar{\mathsf{f}}, \bar{\mathsf{r}}, \mathsf{t}]\!]\}$$

- **Sound:** $\forall(\bar{\mathsf{f}}, \bar{\mathsf{r}}) \in \overline{\Psi}_{pre} \times \overline{\Omega}_{pre} : \alpha_f(\mathbb{S}^{\parallel})[\![\bar{\mathsf{f}}, \bar{\mathsf{r}}]\!] \sqsubseteq_f \overline{\mathbb{S}^{\parallel}}[\![\bar{\mathsf{f}}, \bar{\mathsf{r}}]\!]$

The Example

Thread 1	Thread 2
`a.i=1;` `a.j=1;`	`if(a.j==1 && a.i==0)` `throw new Exception();`

- 1st iteration of the multithread semantics
  - > *visible*: `i=[0..0]` `j=[0..0]`

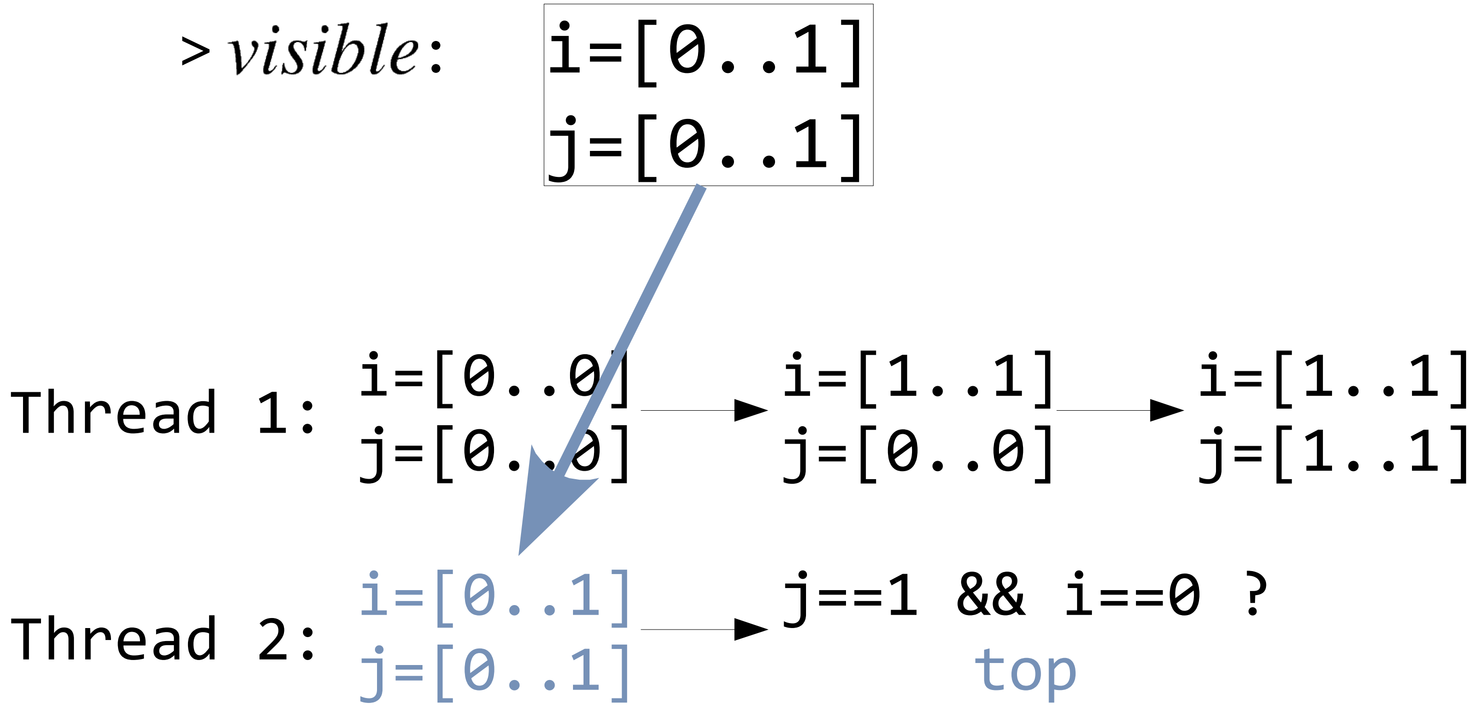


```
Thread 1: i=[0..0]  → i=[1..1]  → i=[1..1]
          j=[0..0]    j=[0..0]    j=[1..1]

Thread 2: i=[0..0]  → j==1 && i==0 ?
          j=[0..0]         false
```

The Example

Thread 1	Thread 2
`a.i=1;` `a.j=1;`	`if(a.j==1 && a.i==0)` `throw new Exception();`

- 2^{nd} iteration of the multithread semantics
  - *visible*: `i=[0..1]` `j=[0..1]`



Thread 1: `i=[0..0]` `j=[0..0]` → `i=[1..1]` `j=[0..0]` → `i=[1..1]` `j=[1..1]`

Thread 2: `i=[0..1]` `j=[0..1]` → `j==1 && i==0 ?` top

The Example

Thread 1	**Thread 2**
`a.i=1;` `a.j=1;`	`if(a.j==1 && a.i==0)` `  throw new Exception();`

- 2nd iteration of the multithread semantics
  - > *visible*: `i=[0..1]` `j=[0..1]`

Thread 1: `i=[0.` `..1]`

The program may throw the exception

Thread 2: `i=[0..` `j=[0..1]` `top`

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

Deposit1	Deposit2

`int t1=a.amount;`
`t1=t1+1000$;`
`a.amount=t1;`
	`int t2=a.amount;`
	`t2=t2+1000$;`
	`a.amount=t2;`

- 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

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

- 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

$$ds(\mathsf{s}_1, \mathsf{s}_2) = \mathsf{false}$$
$$\Updownarrow$$
$$\exists \mathsf{var} \in dom(\mathsf{s}_1) \cap dom(\mathsf{s}_2) : \mathsf{s}_1(\mathsf{var}) = (\mathsf{val}_1, \mathsf{t}_1),$$
$$\mathsf{s}_2(\mathsf{var}) = (\mathsf{val}_2, \mathsf{t}_2), \mathsf{t}_1 \neq \mathsf{t}_2$$

An example

Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	10.000$	System

→

Obj.	Field	Value	Thread
a	amount	11.000$	Deposit1

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	11.000$	Deposit1

→

Obj.	Field	Value	Thread
a	amount	12.000$	Deposit2

Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	10.000$	System

→

Obj.	Field	Value	Thread
a	amount	11.000$	Deposit1

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	10.000$	System

→

Obj.	Field	Value	Thread
a	amount	11.000$	Deposit2

An example

Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	10.000$	System

→

Obj.	Field	Value	Thread
a	amount	11.000$	Deposit1

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	11.000$	Deposit1

→

Obj.	Field	Value	Thread
a	amount	12.000$	Deposit2

Thread Deposit

Obj			
a	amoun		

		alue	Thread
	nt	11.000$	Deposit1

Non-deterministic executions!

Thread Deposit2:

Obj.	Field	Value	Thread
a	amount	10.000$	System

→

Obj.	Field	Value	Thread
a	amount	11.000$	Deposit2

Levels of abstraction

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

- Second level: $\overline{\mathsf{S}} : [\mathsf{Var} \rightarrow (\widehat{\mathsf{V}} \times \wp(\mathsf{TId}))]$
- Trace
  - One abstract value
  - The set of threads that may have written it

- Sound

$$\langle \wp(\Psi), \subseteq \rangle \underset{\alpha_\Psi}{\overset{\gamma_\Psi}{\leftrightarrows}} \langle \widehat{\Psi}, \sqsubseteq_{\widehat{\Psi}} \rangle \underset{\alpha_{\widehat{\Psi}}}{\overset{\gamma_{\widehat{\Psi}}}{\leftrightarrows}} \langle \overline{\Psi}, \sqsubseteq_{\overline{\Psi}} \rangle$$

Determinism on abstract states

- First abstraction

$$\widehat{ds}(\widehat{\mathsf{s}}) = \text{false}$$

$$\Updownarrow$$

$$\exists \text{var} \in dom(\widehat{\mathsf{s}}) : |dom(\widehat{\mathsf{s}}(\text{var}))| > 1$$

- Second abstraction

$$\overline{ds}(\overline{\mathsf{s}}) = \text{false} \Leftrightarrow \exists \text{var} \in dom(\overline{\mathsf{s}}) : |\pi_2(\overline{\mathsf{s}}(\text{var}))| > 1$$

- Soundness

$$\forall \theta \in \wp(\Psi) : d(\theta) = \text{false} \Rightarrow \widehat{d}(\alpha_{\Psi}(\theta)) = \text{false}$$

$$\forall \widehat{\mathsf{f}} \in \widehat{\Psi} : \widehat{d}(\widehat{\mathsf{f}}) = \text{false} \Rightarrow \overline{d}(\alpha_{\widehat{\Psi}}(\widehat{\mathsf{f}})) = \text{false}$$

An example – 1st abstraction

Thread Deposit1:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]$

→

Obj.	Field	Thread	Value
a	amount	Deposit1	[11.000..11.000]$

Thread Deposit2:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]$
		Deposit1	[11.000..11.000]$

→

Obj.	Field	Thread	Value
a	amount	Deposit2	[11.000..12.000]$

An example – 1st abstraction

Thread Deposit1:

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]$

→

Obj.	Field	Thread	Value
a	amount	Deposit1	[11.000..11.000]$

Non-deterministic program!

Thread Deposit

Obj.	Field	Thread	Value
a	amount	System	[10.000..10.000]$
		Deposit1	[11.000..11.000]$

→

Obj.	Field	Thread	Value
a	amount	Deposit2	[11.000..12.000]$

Weak determinism

$$\begin{aligned}\widehat{wds} &: [\widehat{\mathsf{S}} \rightarrow \{\mathsf{true}, \mathsf{false}\}] \\ \widehat{wds} &(\widehat{\mathsf{s}}) = \mathsf{false}\end{aligned}$$

$$\Updownarrow$$

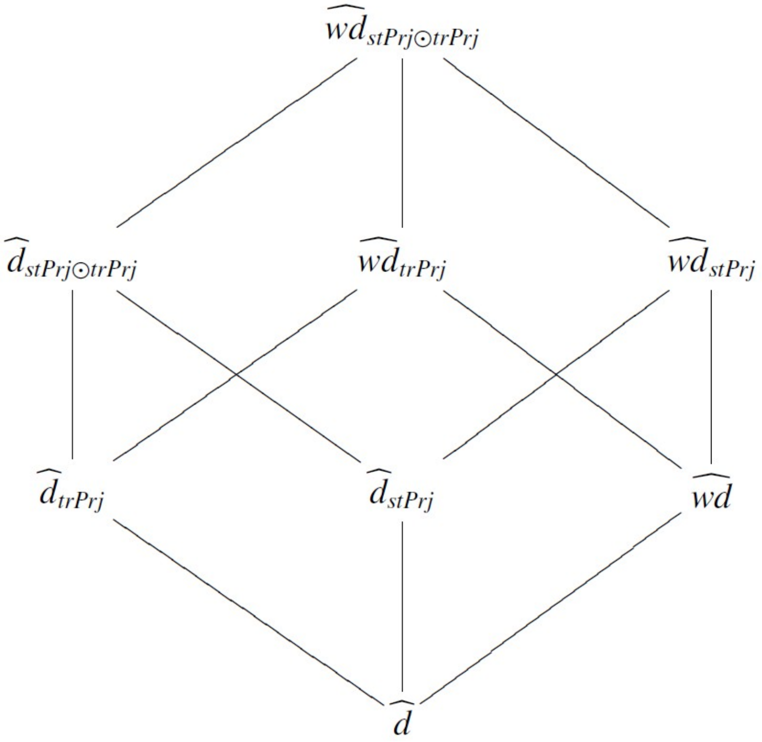
$$\exists \mathsf{var} \in dom(\widehat{\mathsf{s}}) : |dom(\widehat{\mathsf{s}}(\mathsf{var}))| > 1$$

$$\wedge \exists \mathsf{t}_1, \mathsf{t}_2 \in dom(\widehat{\mathsf{s}(\mathsf{var})}) : \widehat{\mathsf{s}}(\mathsf{var})(\mathsf{t}_1) \neq \widehat{\mathsf{s}}(\mathsf{var})(\mathsf{t}_2)$$

- 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

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

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: $\mathsf{Op} = \mathbb{ST}(\mathsf{Val})$
    - Heap: $\mathsf{H} : [\mathsf{Ref} \rightarrow (\mathsf{Obj} \cup \mathsf{Arr} \cup \mathsf{Str})]$
    - Local variables: $\mathsf{LV} = \mathbb{AR}(\mathsf{Val})$
    - Locked monitors: $\mathsf{L} : [\mathsf{Ref} \rightarrow \mathbb{N}]$
- We represent programs as Control Flow Graph CFG
- 1st abstraction:
  - > Executions on the Control Flow Graph exCFG
  - > Sound abstraction of real executions

$$\langle \wp(\Sigma^{\vec{+}}), \subseteq \rangle \underset{\alpha_{\mathsf{CFG}}}{\overset{\gamma_{\mathsf{CFG}}}{\leftrightarrows}} \langle \mathsf{exCFG}, \sqsubseteq_{\mathsf{CFG}} \rangle$$

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: $\overline{\mathsf{Op}} = \mathbb{ST}(\overline{\mathsf{Val}})$
  - Heap: $\overline{\mathsf{H}} : [\overline{\mathsf{P}} \rightarrow (\overline{\mathsf{Obj}} \cup \overline{\mathsf{Arr}} \cup \overline{\mathsf{Str}})]$
  - Local variables: $\overline{\mathsf{LV}} = \mathbb{AR}(\overline{\mathsf{Val}})$
  - Locked monitors: $\overline{\mathsf{L}} : [\overline{\mathsf{Ref}} \rightarrow \mathbb{N}]$
- Proved the soundness

$$\langle \wp(\Sigma), \subseteq \rangle \underset{\alpha_\Sigma}{\overset{\gamma_\Sigma}{\leftrightarrows}} \langle \overline{\Sigma}, \sqsubseteq_\Sigma \rangle$$

- Operational semantics of statements
- Proved the local soundness

$$\forall \sigma \in \Sigma : \alpha_\Sigma(\{\sigma' : \sigma \rightarrow \sigma'\}) \sqsubseteq_\Sigma \overline{\sigma}' : \alpha_\Sigma(\{\sigma\}) \overline{\rightarrow} \overline{\sigma}'$$

- Applied to HBMM and determinism

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

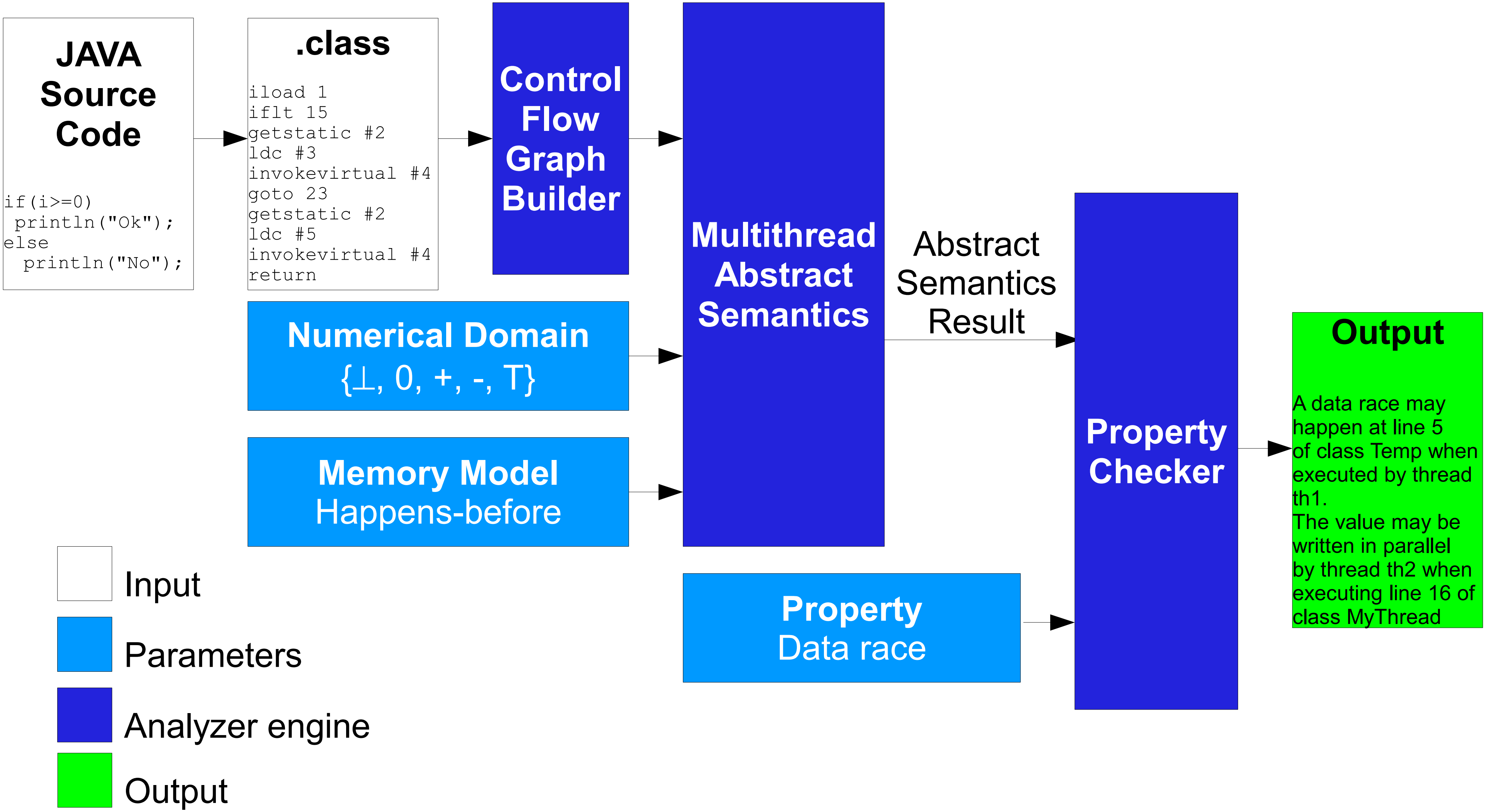
6. Static analysis of unsafe code
   - An industrial application of generic analyzer

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

Architecture



JAVA Source Code

```
if(i>=0)
 println("Ok");
else
  println("No");
```

.class

```
iload 1
iflt 15
getstatic #2
ldc #3
invokevirtual #4
goto 23
getstatic #2
ldc #5
invokevirtual #4
return
```

Control Flow Graph Builder

Numerical Domain
{⊥, 0, +, -, T}

Memory Model
Happens-before

Multithread Abstract Semantics

Abstract Semantics Result

Property
Data race

Property Checker

Output

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

- ☐ Input
- 🟦 Parameters
- 🟦 Analyzer engine
- 🟩 Output

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

[JMM] J. Manson, W. Pugh, and S. V. Adve. *The Java memory model*. In ACM Press, editor, Proceedings of POPL ’05, 2005.
[LEA] D. Lea. Concurrent Programming in Java. Addison-Wesley, 1996.
[PRA] C. Von Praun and T. R. Gross. *Object race detection*. In ACM Press, editor, Proceedings of OOPSLA 01, 2001.
[BENCH] *Java Grande Forum Benchmark Suite*. At http://www.epcc.ed.ac.uk/research/activities/java-grande/

External benchmarks

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

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

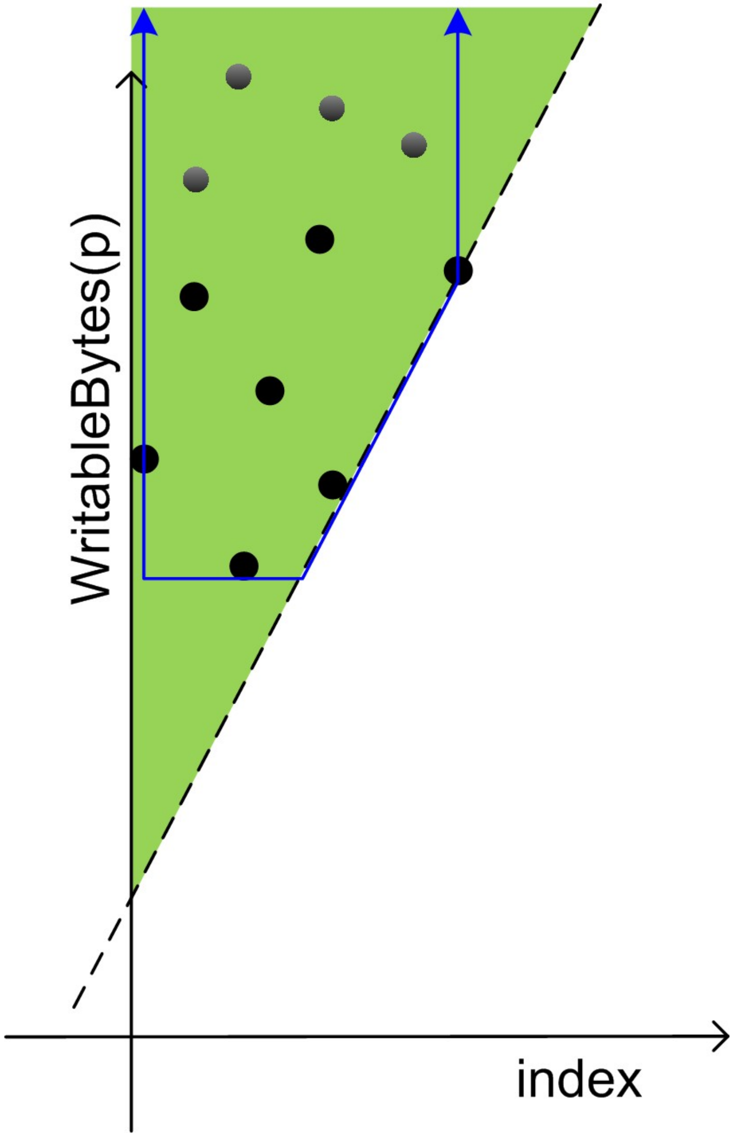
Generic analyzers

- 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

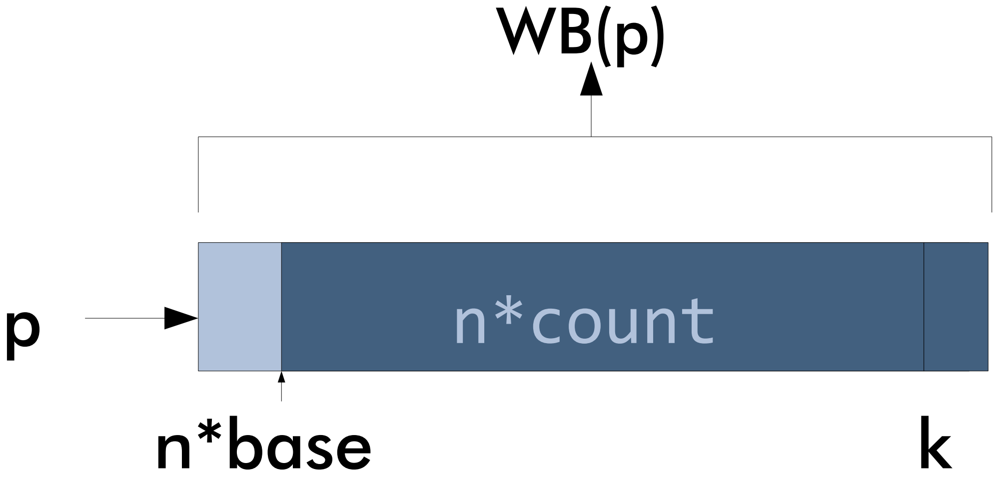
- .Net: safe environment of execution
  - Exception: unsafe code
  - Direct access to the memory
- No guarantees on direct memory accesses
  - Buffer overrun
- Goal: apply Clousot to unsafe code
- Combination with contracts
  - Method boundary annotation
  - Lightweight domains
    - Stripes: new relational domain
    - Combined with well-known numerical domains
  - Scalability
    - .Net libraries analyzed in a couple of minutes

Numerical Domains - Stripes



$$\mathtt{WB(p)} \geq \mathtt{n} * (\mathtt{count}[+\mathtt{base}]) + \mathtt{k}$$

- Relational
- Precise



- Linear complexity in practice

Example: `InitToZero`

$$\frac{\begin{array}{l}\text{check}(\text{WB}(\text{p}) \geq \text{sizeof}(\text{x}) + \text{exp} * \text{sizeof}(*\text{p}), \bar{\text{s}}) = \text{true} \\ \text{check}(\text{exp} \geq 0, \bar{\text{s}}) = \text{true}\end{array}}{\mathbb{F}[\![*(\text{p} + \text{exp}) = \text{x}]\!](\bar{\text{s}}) \rightarrow \bar{\text{s}}}$$

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

Example: `InitToZero`

$$\frac{\begin{array}{l}\mathsf{check}(\mathsf{WB}(\mathsf{p}) \geq \mathsf{sizeof}(\mathsf{x}) + \mathsf{exp} * \mathsf{sizeof}(*\mathsf{p}), \bar{\mathsf{s}}) = \mathsf{true} \\ \mathsf{check}(\mathsf{exp} \geq 0, \bar{\mathsf{s}}) = \mathsf{true}\end{array}}{\mathbb{F}[\![*(\mathsf{p} + \mathsf{exp}) = \mathsf{x}]\!](\bar{\mathsf{s}}) \rightarrow \bar{\mathsf{s}}}$$

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

Infer:

```
WB(ptr)≥4*len
len≥i+1
WB(ptr)≥4*i+4
```

Example: `InitToZero`

$$\frac{\begin{array}{l}\text{check}(\text{WB}(\text{p}) \geq \text{sizeof}(\text{x}) + \text{exp} * \text{sizeof}(*\text{p}), \bar{\text{s}}) = \text{true} \\ \text{check}(\text{exp} \geq 0, \bar{\text{s}}) = \text{true}\end{array}}{\mathbb{F}[\![*(\text{p} + \text{exp}) = \text{x}]\!](\bar{\text{s}}) \rightarrow \bar{\text{s}}}$$

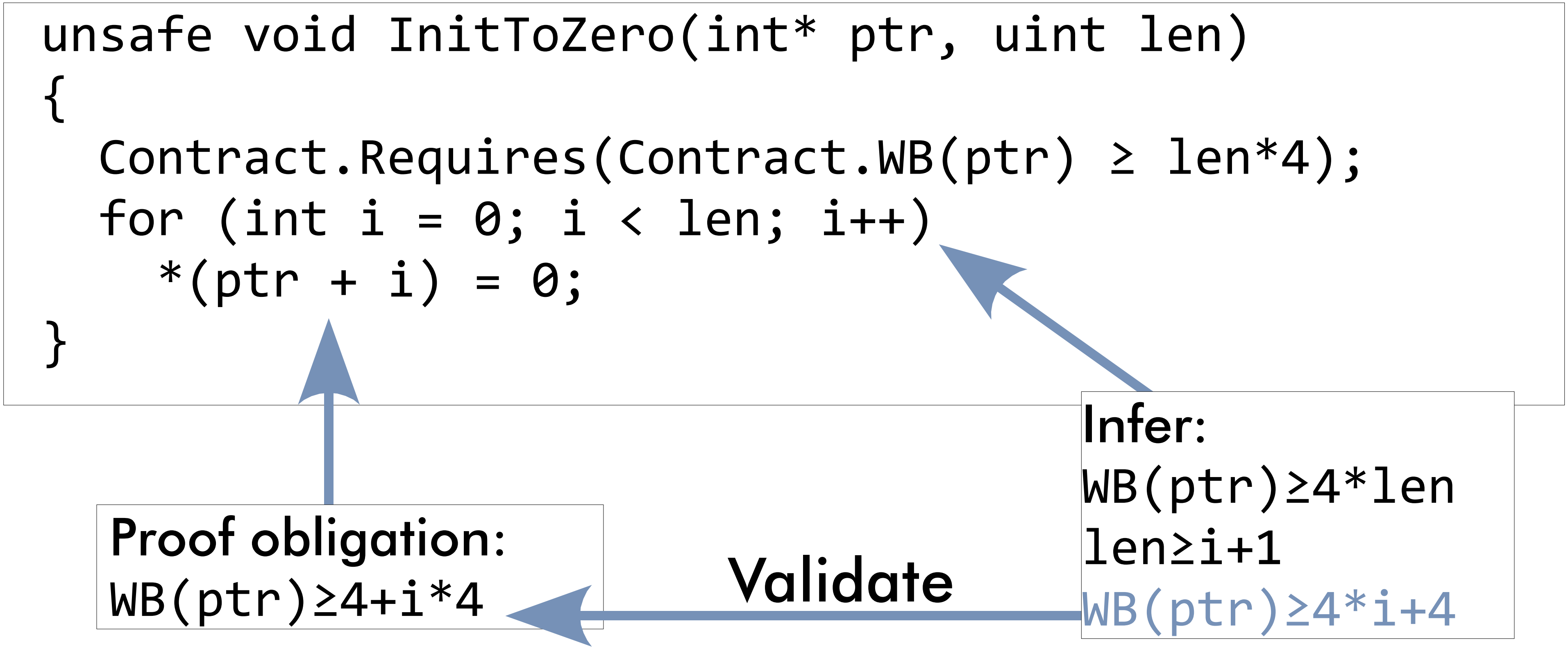
```
unsafe void InitToZero(int* ptr, uint len)
{
  Contract.Requires(Contract.WB(ptr) ≥ len*4);
  for (int i = 0; i < len; i++)
    *(ptr + i) = 0;
}
```

Proof obligation:
`WB(ptr)≥4+i*4`

Infer:
`WB(ptr)≥4*len`
`len≥i+1`
`WB(ptr)≥4*i+4`

Example: `InitToZero`

$$\frac{\begin{array}{l}\mathsf{check}(\mathsf{WB}(\mathsf{p}) \geq \mathsf{sizeof}(\mathsf{x}) + \mathsf{exp} * \mathsf{sizeof}(*\mathsf{p}), \bar{\mathsf{s}}) = \mathsf{true} \\ \mathsf{check}(\mathsf{exp} \geq 0, \bar{\mathsf{s}}) = \mathsf{true}\end{array}}{\mathbb{F}[\![*(\mathsf{p} + \mathsf{exp}) = \mathsf{x}]\!](\bar{\mathsf{s}}) \rightarrow \bar{\mathsf{s}}}$$



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

Infer:
`WB(ptr)≥4*len`
`len≥i+1`
`WB(ptr)≥4*i+4`

Validate

Proof obligation:
`WB(ptr)≥4+i*4`

Intervals

$$\frac{\begin{array}{l}\text{check}(\text{WB}(\text{p}) \geq \text{sizeof}(\text{x}) + \text{exp} * \text{sizeof}(*\text{p}), \bar{\text{s}}) = \text{true} \\ \text{check}(\text{exp} \geq 0, \bar{\text{s}}) = \text{true}\end{array}}{\mathbb{F}[\![*(\text{p} + \text{exp}) = \text{x}]\!](\bar{\text{s}}) \to \bar{\text{s}}}$$

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

```
WB(ptr)≥4*len
len≥i+1
WB(ptr)≥4*i+4
```

- **Stripes do not prove i≥0**

Intervals

$$\frac{\begin{array}{l}\text{check}(\text{WB}(\text{p}) \geq \text{sizeof}(\text{x}) + \text{exp} * \text{sizeof}(*\text{p}), \bar{\text{s}}) = \text{true}\\ \text{check}(\text{exp} \geq 0, \bar{\text{s}}) = \text{true}\end{array}}{\mathbb{F}[\![*(\text{p} + \text{exp}) = \text{x}]\!](\bar{\text{s}}) \rightarrow \bar{\text{s}}}$$

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

```
WB(ptr)≥4*len
len≥i+1
WB(ptr)≥4*i+4
```

- Stripes do not prove i≥0

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

```
i=[0..+∞]
```

- Intervals do it!

Experimental Results

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

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

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

[Ferr08] P. Ferrara, "*Static analysis via abstract interpretation of the happens-before memory model*", in Springer editor, Proceedings of TAP 2008, volume 4966 of LNCS, Prato, Italy, April 9-11, 2008

[Ferr08b] P. Ferrara, "*Static analysis of the determinism of multithreaded programs*", in IEEE Computer Society, editor, Proceedings of SEFM 2008, Cape Town, South Africa, November 10-14, 2008

[Ferr08a] P. Ferrara, "*A fast and precise analysis for data race detection*", in Elsevier editor, Proceedings of Bytecode'08, ENTCS, Budapest, Hungary, April 6, 2008

[FLF08] P. Ferrara, F. Logozzo and M. Fähndrich, "*Safer unsafe code for .NET*", in ACM Press, editor, Proceedings of OOPSLA 2008, Nashville, USA, October 19-23, 2008

Question time

Thank you!