

# Static analysis by abstract interpretation of Java multithreaded programs

Pietro Ferrara

École Normale Supérieure  
Paris, France  
Università Ca' Foscari  
Venice, Italy

Chair of Programming Methodology, ETH, Zurich, Switzerland

# Multicore revolution

- The only way to extend **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.

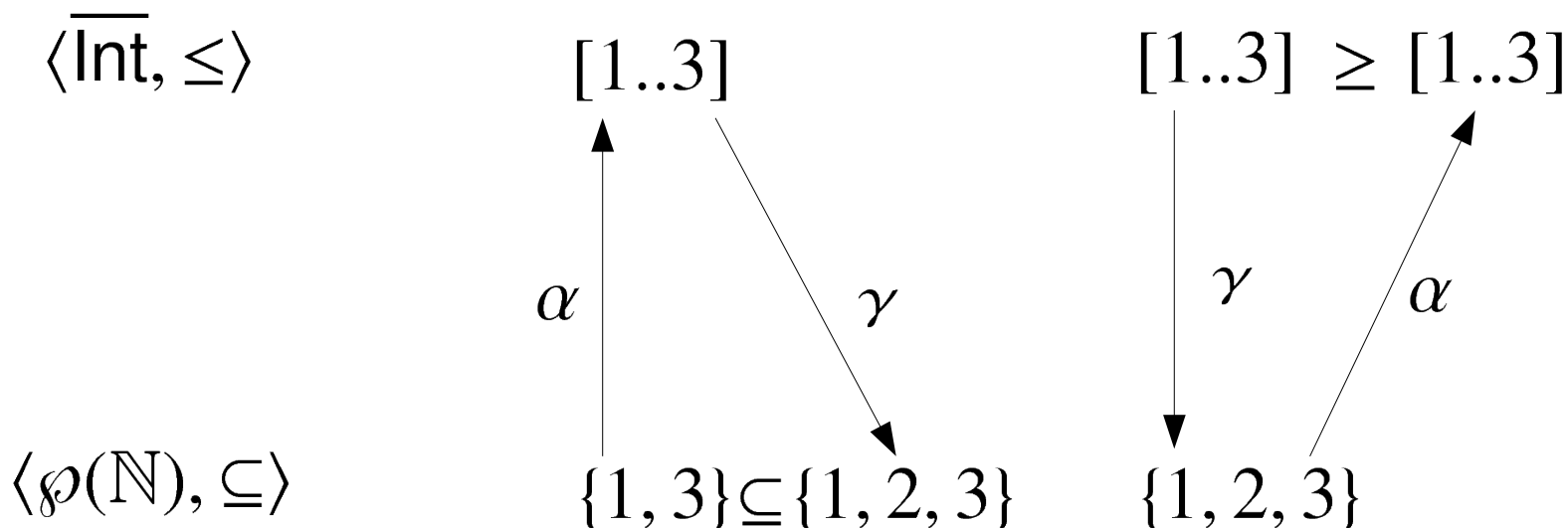
- Parallelism supported through **multithreading**
  - > Java
  - > C#
- Implicit communications via shared memory
- Synchronization on monitors
- **Subtle** and problematic

# Static analysis

- Infer and prove properties
  - > at **compile time**
  - > respected by **all possible executions**
- Tradeoff between precision and efficiency
- **Abstract interpretation**
  - > Mathematical theory developed by P. & R. Cousot
  - > **Define** semantics of programs
  - > **Soundly approximate** it
- **Main components:**
  - > **Semantics** of the program
  - > **Domain**
  - > **Property** of interest

# Domain

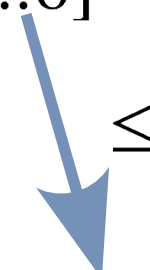
- Concrete: runtime behaviors, not computable
- Abstract: computable approximation
- Abstraction and concretization functions
  - > Soundness formally proved
    - Domains: **Galois connection**
- E.g.: integer values abstracted with intervals



# Semantics

- Concrete  $\mathbb{S}$  and abstract  $\bar{\mathbb{S}}$
- Soundness:  $\forall c \in C : \alpha(\mathbb{S}[c]) \leq \bar{\mathbb{S}}[\alpha(c)]$
- E.g., sum using intervals

$$\alpha(\{1, 3\} + \{1, 3\}) = \alpha(\{2, 4, 6\}) = [2..6]$$

$$\alpha(\{1, 3\}) \oplus \alpha(\{1, 3\}) = [1..3] \oplus [1..3] = [2..6]$$


- Usually semantics defined in **fixpoint** form
  - > Small step semantics of statements
    - Locally sound
  - > Fixpoint computes all the executions

# Outline

1. Introduction
2. Happens-before memory model
  - Definition in fixpoint form and abstraction
3. Determinism of multithreaded programs
  - New property
4. Domain and semantics of Java bytecode
  - Low-level domain, specific alias analysis
5. Checkmate
  - 1<sup>st</sup> generic 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
- Java MM introduced in 2005, runtime informations
- Happens-before MM (HBMM) – L. Lamport 1978
  - > Over-approximation of the Java MM
- Main components:
  - > Program order (intra-thread order of statements)
  - > Synchronizes-with relation
- Happens-before order: a1 happens-before a2 if
  - > a1 appears before a2 in the program order
  - > a2 synchronizes-with a1
  - > You can reach a2 starting from a1

# Consistency rule

- **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:
    - $r$  does not hb  $w$
    - There is no  $w'$  to  $v$  that hb  $r$  and such that  $w$  hb it
- **We focus on**
  - > Mutual exclusion
  - > Launch of a thread

# An example

Thread 1	Thread 2
<code>i=1;</code> <code>j=1;</code>	<code>if(j==1 &amp;&amp; i==0)</code> <code>throw new Exception();</code>

- May the **exception** be thrown?
  - > Sequential consistency: no!
  - > HBMM e Java MM: **yes!**
- E.g. **swap** of independent statements

# Concrete domain and semantics

- **Generic** w.r.t. programming language
  - > Parameterized on semantics of statements
- **Collect** for each thread its trace of execution
$$\Psi : TId \rightarrow St^{\vec{t}}$$
- **Abstract** the inter-thread order of execution
- **The semantics computes all possible executions**
$$\langle \wp(\Psi), \subseteq, \emptyset, \Psi, \cup, \cap \rangle$$
- **Intuition**
  - > Given a multithreaded execution
  - > Reading from shared memory take one of
    - **Visible values** of the multithreaded execution
    - Following the consistency rule
- **Two nested fixpoints**

# Concrete semantics

- Intra-thread semantics

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

$$\mathbb{S}^\circ \llbracket f, r, t \rrbracket = \text{lfp}_{\emptyset}^{\subseteq} \lambda T. \{\sigma_0\} \cup \{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \\ \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \text{step}(t, f, r)\}$$

- Multithread semantics

$$\mathbb{S}^{\parallel} : \Psi \times \Omega \mapsto \wp(\Psi \times \Omega)$$

$$\mathbb{S}^{\parallel} \llbracket f_0, r_0 \rrbracket = \text{lfp}_{\emptyset}^{\subseteq} \lambda \Phi. \{(f_0, r_0)\} \cup \{(f_i, r) : \exists (f_{i-1}, r) \in \Phi : \\ \forall t \in \text{dom}(f_{i-1}) : f_i(t) \in \mathbb{S}^\circ \llbracket f_{i-1}, r, t \rrbracket, \\ f_i(t) = \sigma_0 \rightarrow \dots \rightarrow \sigma_i, \sigma_i \in \text{St}_{\rightarrow}^{\circ}\}$$

# The Example

Thread 1	Thread 2
<code>i=1;</code> <code>j=1;</code>	<code>if(j==1 &amp;&amp; i==0)</code> <code>throw new Exception();</code>

- 1<sup>st</sup> iteration of multithread semantics

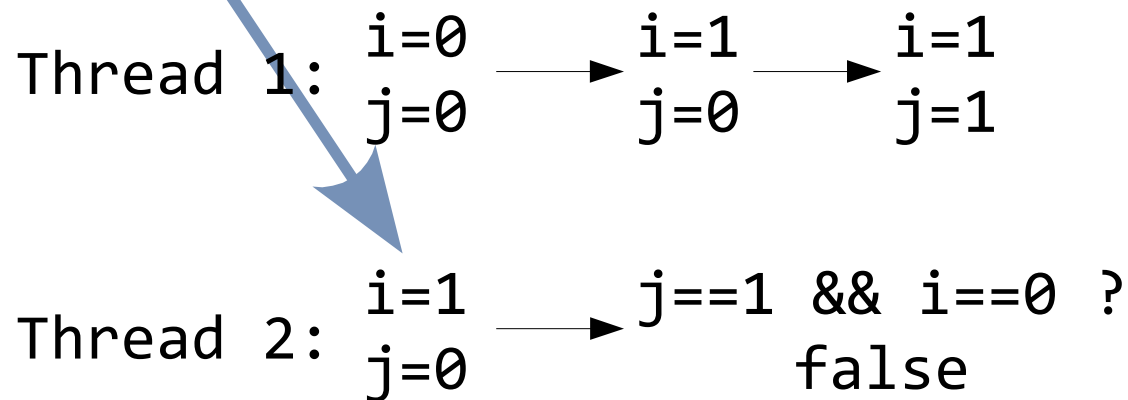
Thread 1:  $i=0$   $\longrightarrow$   $i=1$   $\longrightarrow$   $i=1$   
 $j=0$   $\longrightarrow$   $j=0$   $\longrightarrow$   $j=1$

Thread 2:  $i=0$   $\longrightarrow$   $j==1 \ \&\& \ i==0 \ ?$   
 $j=0$   $\longrightarrow$   $\text{false}$

# The Example

Thread 1	Thread 2
<code>i=1;</code> <code>j=1;</code>	<code>if(j==1 &amp;&amp; i==0)</code> <code>throw new Exception();</code>

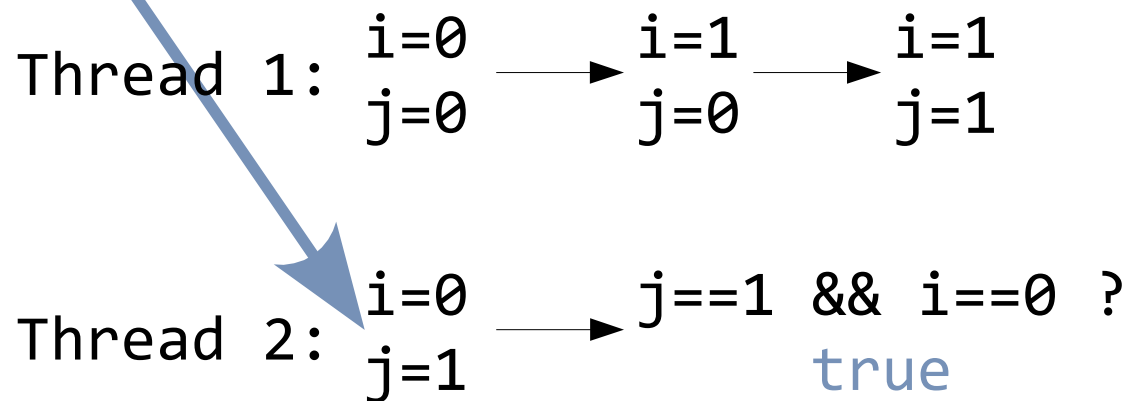
- 2<sup>nd</sup> iteration of multithread semantics
  - > Previous iteration: Thread 1 writes in parallel
    - `i=1`
    - `j=1`



# The Example

Thread 1	Thread 2
<code>i=1;</code> <code>j=1;</code>	<code>if(j==1 &amp;&amp; i==0)</code> <code>throw new Exception();</code>

- 2<sup>nd</sup> iteration of multithread semantics
  - > Previous iteration: Thread 1 writes in parallel
    - `i=1`
    - `j=1`



# Abstract domain and semantics

- Straight abstraction of the concrete one:

$$\bar{\Psi} : \text{TId} \rightarrow \overline{\text{St}}^{\vec{f}}$$

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

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

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

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

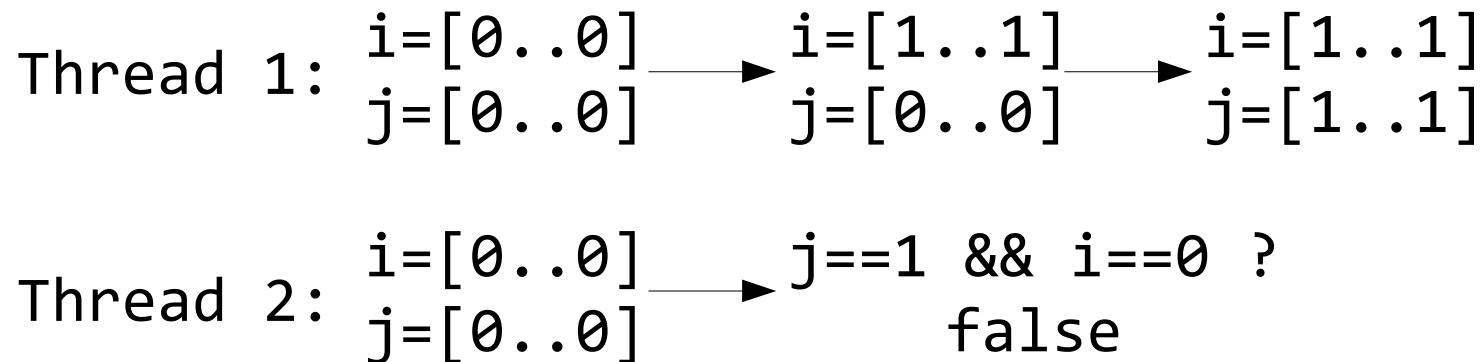
$$\bar{\mathbb{S}}^{\parallel} [\bar{f}_0, \bar{r}_0] = \text{lfp}_{\emptyset}^{\sqsubseteq_f} \lambda (\bar{f}, \bar{r}). \{(\bar{f}_0, \bar{r}_0)\} \sqcup_f \{(\bar{f}_i, \bar{r}) : \forall t \in \text{dom}(\bar{f}) : \bar{f}_i(t) = \bar{\mathbb{S}}^{\circ} [\bar{f}, \bar{r}, t]\}$$

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

# The Example

Thread 1	Thread 2
<code>i=1;</code> <code>j=1;</code>	<code>if(j==1 &amp;&amp; i==0)</code> <code>throw new Exception();</code>

- 1<sup>st</sup> iteration of multithread semantics



# The Example

Thread 1	Thread 2
<code>i=1;</code> <code>j=1;</code>	<code>if(j==1 &amp;&amp; i==0)</code> <code>throw new Exception();</code>

- 2<sup>nd</sup> iteration of multithread semantics
  - > Previous iteration: Thread 1 writes in parallel
    - `i=[1..1]`
    - `j=[1..1]`

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

Thread 2: `i=[0..1]` → `j==1 && 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
  - New property
4. Domain and semantics of Java bytecode
  - Low-level domain, specific alias analysis
5. Checkmate
  - 1<sup>st</sup> generic analyzer of multithreaded programs
6. Static analysis of unsafe code
  - An industrial application of generic analyzer

# An example

## Thread Deposit1

Variable	Value
tempt1	
...	




```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	
...	



```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```

# An example

## Thread Deposit1

Variable	Value
tempt1	10.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	
...	


```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```

# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```




## Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```



# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	
...	


```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```

# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```



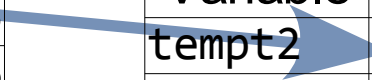

## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```




# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```




## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	12.000\$
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```



# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	12.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	12.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```

# An example


## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```



## Shared Memory

Object	Field	Value
a	amount	12.000\$
	ID	58656
	...	
...		


## Thread Deposit2

Variable	Value
tempt2	12.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```




At the end:

**a.amount= 12.000\$**

# An example

## Thread Deposit1

Variable	Value
tempt1	
...	




```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	
...	



```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```

# An example

## Thread Deposit1

Variable	Value
tempt1	10.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```

# An example

## Thread Deposit1

Variable	Value
tempt1	10.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
  
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	10.000\$
...	


```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
  
a.amount=tempt2;
```

# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```




## Shared Memory

Object	Field	Value
a	amount	10.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	10.000\$
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```



# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	10.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```


```
a.amount=tempt2;
```

# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;  
tempt1=tempt1+1000$;  
a.amount=tempt1;
```




## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;  
tempt2=tempt2+1000$;  
a.amount=tempt2;
```



# An example


## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

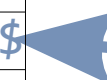


## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2


Variable	Value
tempt2	11.000\$
...	



```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```



# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```

At the end:

**a.amount=11.000\$**

# An example

## Thread Deposit1

Variable	Value
tempt1	11.000\$
...	

```
int tempt1=a.amount;
```

```
tempt1=tempt1+1000$;
```

```
a.amount=tempt1;
```

## Shared Memory

Object	Field	Value
a	amount	11.000\$
	ID	58656
	...	
...		

## Thread Deposit2

Variable	Value
tempt2	11.000\$
...	

```
int tempt2=a.amount;
```

```
tempt2=tempt2+1000$;
```

```
a.amount=tempt2;
```

1<sup>st</sup> execution: a.amount=12.000\$  
2<sup>nd</sup> execution: a.amount=11.000\$

# 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 random interleaving
  - > **Flexible**

# Concrete domain

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

- Each value related to a thread identifier
    - > Trace **which thread wrote** it in the shared memory
  - **Thread-partitioned** trace domain
    - > Relates each thread to its trace of execution
      - Same approach of HBMM
- $$\Psi : [\text{TId} \rightarrow S^{\vec{t}}]$$
- **Concrete semantics: set of functions**
    - > All the possible executions

# Determinism on concrete states

$$ds(s_1, s_2) = \text{false}$$



$$\exists \text{var} \in \text{dom}(s_1) \cap \text{dom}(s_2) : s_1(\text{var}) = (\text{val}_1, t_1),$$

$$s_2(\text{var}) = (\text{val}_2, t_2), t_1 \neq t_2$$

- Difference among concrete executions
- 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 that relate
    - the **same variable**
    - to **values written by different threads**

# An example – Concrete semantics

## Thread Deposit1:

Obj.	Field	Value	Thread
a	amount	10.000\$	System



Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1

## Thread Deposit2:

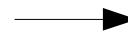
Obj.	Field	Value	Thread
a	amount	11.000\$	Deposit1



Obj.	Field	Value	Thread
a	amount	12.000\$	Deposit2

## Thread Deposit1:

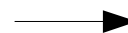
Obj.	Field	Value	Thread
a	amount	10.000\$	System



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

# First and second abstraction

- First level:  $\widehat{S} : [\text{Var} \rightarrow [\text{TId} \rightarrow \widehat{V}]]$
- Parameterized on an abstract domain  $\widehat{V}$ 
  - > One value for each thread
- Second level:  $\overline{S} : [\text{Var} \rightarrow (\widehat{V} \times \wp(\text{TId}))]$
- Trace
  - > One abstract value
  - > The set of threads that may have written it
- Sound

$$\langle \wp(\Psi), \sqsubseteq \rangle \xrightleftharpoons[\alpha_\Psi]{\gamma_\Psi} \langle \widehat{\Psi}, \sqsubseteq_{\widehat{\Psi}} \rangle \xrightleftharpoons[\alpha_{\widehat{\Psi}}]{\gamma_{\widehat{\Psi}}} \langle \overline{\Psi}, \sqsubseteq_{\overline{\Psi}} \rangle$$

# Determinism on abstract states

- First abstraction

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



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

- Second abstraction

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

- Soundness

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

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

# An example – 1<sup>st</sup> 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 – 2<sup>nd</sup> abstraction

## Thread Deposit1:

Obj.	Field	Value	Threads
a	amount	[10.000..10.000]\$	{System}



Obj.	Field	Value	Threads
a	amount	[11.000..11.000]\$	{Deposit1}

## Thread Deposit2:

Obj.	Field	Value	Threads
a	amount	[10.000..11.000]\$	{System, Deposit1}



Obj.	Field	Value	Threads
a	amount	[11.000..12.000]\$	{Deposit2}

# Weak determinism

$$\widehat{wds} : [\widehat{S} \rightarrow \{\text{true}, \text{false}\}]$$

$$\widehat{wds}(\widehat{s}) = \text{false}$$



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

$$\wedge \exists t_1, t_2 \in \text{dom}(\widehat{s}(\text{var})) : \widehat{s}(\text{var})(t_1) \neq \widehat{s}(\text{var})(t_2)$$

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

# An example – 1<sup>st</sup> 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 – 1<sup>st</sup> abstraction

## Thread Deposit1:

Obj.	Field	Thread	Value
a	amount	System	+



Obj.	Field	Thread	Value
a	amount	Deposit1	+

## Thread Deposit2:

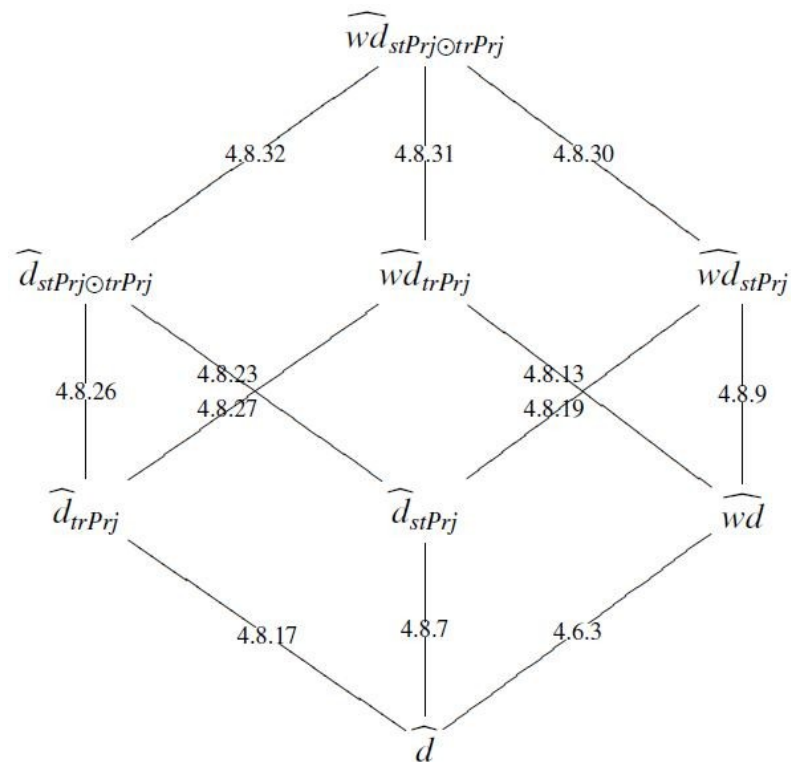
Obj.	Field	Thread	Value
a	amount	System	+
		Deposit1	+



Obj.	Field	Thread	Value
a	amount	Deposit2	+

# Projecting states and traces

- Check the determinism only
  - > On a subset of the shared variables, e.g.
    - Only the amount of the bank account
  - > On a subset of the trace
    - Only the actions that deposit or withdraw money



# Semi-automatic parallelization

- Given a sequential program
  - > **Divide** it in two partitions
  - > **Analyze** them as executed in parallel
  - > **Check** if the determinism is respected
    - Or one of its relaxations

# Semi-automatic parallelization

- Given a sequential program
  - > Divide it in two partitions
  - > Analyze them as executed in parallel
  - > Check if the determinism is respected
    - Or one of its relaxations

Thread 1 → `acc.deposit(1.000$);`  
`acc.deposit(1.000$);` ← Thread 2

Non-determinism on `acc.amount`

# Semi-automatic parallelization

- Given a sequential program
  - > **Divide** it in two partitions
  - > **Analyze** them as executed in parallel
  - > **Check** if the determinism is respected
    - Or one of its relaxations

Thread 1 → `acc.deposit(1.000$);`  
`acc.printAmount();` ← Thread 2

Non-determinism on stdout

# Outline

1. Introduction
2. ~~Happens-before memory model~~
  - ~~Definition in fixpoint form and abstraction~~
3. ~~Determinism of multithreaded programs~~
  - ~~New property~~
4. Domain and semantics of Java bytecode
  - Low-level domain, specific alias analysis
5. Checkmate
  - 1<sup>st</sup> generic 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
- Whole program analysis
- We support all the Java bytecode language

# Concrete domain

- Low-level domain
- Simulate the specification of the JVM
  - > Operand stack  $Op = \mathcal{ST}(\text{Val})$
  - > Local variables  $LV = \mathcal{AR}(\text{Val})$
  - > Object  $Obj : [(C \times F) \rightarrow \text{Val}]$
  - > Heap  $H : [\text{Ref} \rightarrow (\text{Obj} \cup \text{Arr} \cup \text{Str})]$
  - > Monitor  $L : [\text{Ref} \rightarrow \mathbb{N}]$
  - > State  $\Sigma = Op \times LV \times H \times L$
- We represent programs as **CFG**
- **Sound** abstraction of real executions

$$\langle \wp(\Sigma^{\vec{\tau}}), \subseteq \rangle \begin{array}{c} \xleftarrow{\gamma_{\text{CFG}}} \\ \xrightarrow{\alpha_{\text{CFG}}} \end{array} \langle \text{CFG}, \sqsubseteq_{\text{CFG}} \rangle$$

# Alias analysis

- We need to check when references
  - > Always point to the same location (**must-aliasing**)
  - > May point to the same location (**may-aliasing**)
- Concrete references: potentially **infinite**
- May aliasing:
  - > Approximate it with a **finite set**
  - > Idea: binded to the **program point** that allocates it
- Must aliasing:
  - > Set of **equivalence classes**
  - > Each reference related to an equivalence class
  - > **Rough** but precise enough
    - **Trivial** equivalences

# Abstract domain

- Other components: (almost) straight abstraction

- > Operand stack  $\overline{\text{Op}} = \text{ST}(\overline{\text{Val}})$
- > Local variables  $\overline{\text{LV}} = \text{AR}(\overline{\text{Val}})$
- > Object  $\overline{\text{Obj}} = (\text{C} \times \text{F}) \rightarrow \overline{\text{Val}}$
- > Heap  $\overline{\text{H}} = \overline{\text{P}} \mapsto (\overline{\text{Obj}} \cup \overline{\text{Arr}} \cup \overline{\text{Str}})$
- > Monitor  $\overline{\text{L}} : [\text{Ref} \rightarrow \mathbb{N}]$
- > State  $\overline{\Sigma} = \overline{\text{Op}} \times \overline{\text{LV}} \times \overline{\text{H}} \times \overline{\text{L}}$

- Proved the local **soundness**

$$\langle \wp(\Sigma), \sqsubseteq \rangle \begin{matrix} \xleftarrow{\gamma_\Sigma} \\ \xrightarrow{\alpha_\Sigma} \end{matrix} \langle \overline{\Sigma}, \sqsubseteq_\Sigma \rangle$$

# Concrete and abstract semantics

- Operational semantics of statements

$$\frac{os' = push(os, lv[i])}{\langle load \#i, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv, h, l, next(pc) \rangle}$$
$$\frac{\overline{os'} = push(\overline{os}, \overline{lv}[i])}{\langle load \#i, (\overline{os}, \overline{lv}, \overline{h}, \overline{l}, pc) \rangle \overrightarrow{\langle \overline{os'}, \overline{lv}, \overline{h}, \overline{l}, next(pc) \rangle}}$$

- Proved the **local soundness**

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

- Applied to HBMM and determinism

# Outline

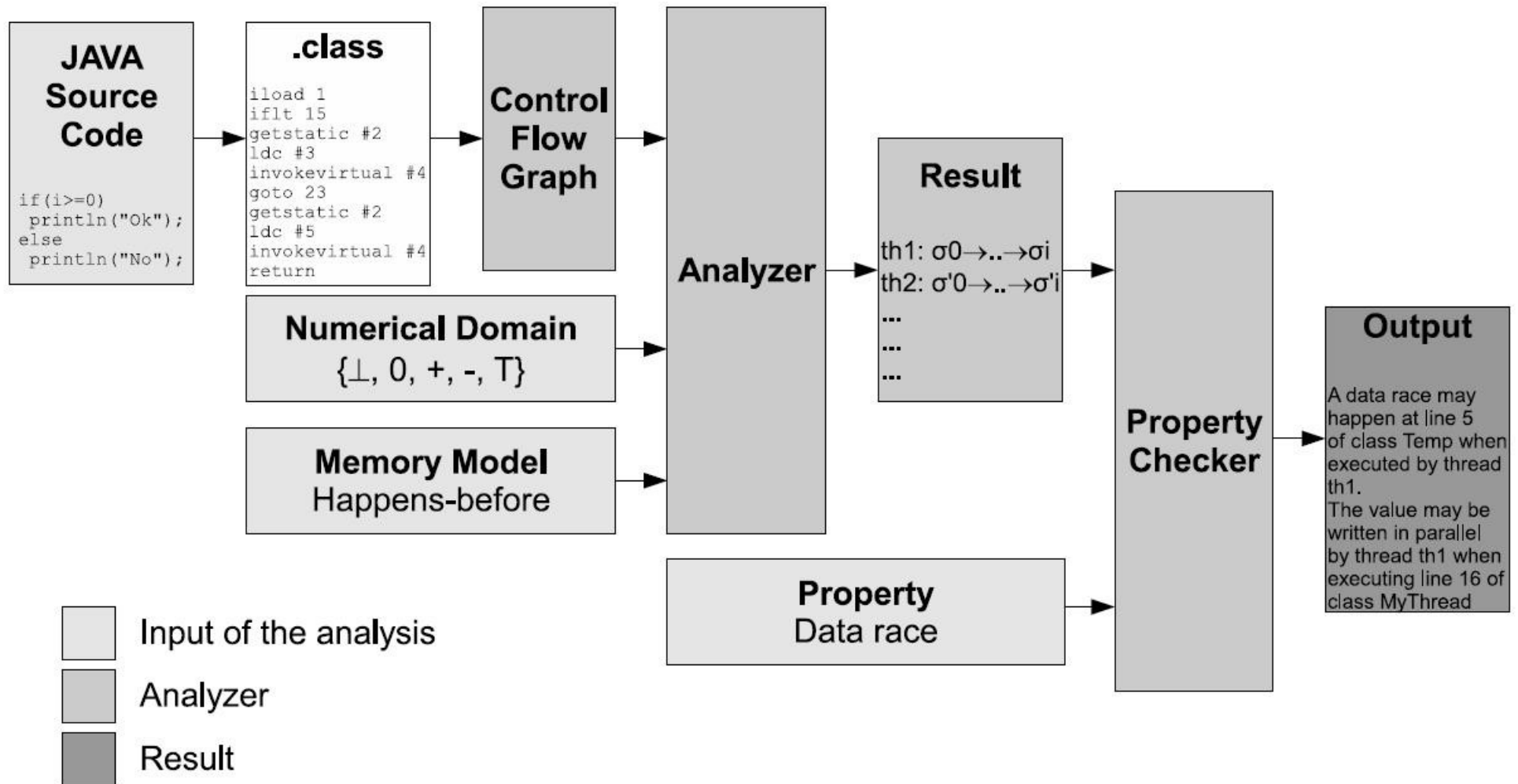
1. Introduction
2. ~~Happens-before memory model~~
  - ~~Definition in fixpoint form and abstraction~~
3. ~~Determinism of multithreaded programs~~
  - ~~New property~~
4. ~~Domain and semantics of Java bytecode~~
  - ~~Low-level domain, specific alias analysis~~
5. Checkmate
  - 1<sup>st</sup> generic analyzer of multithreaded programs
6. Static analysis of unsafe code
  - An industrial application of generic analyzer

# Checkmate

- 1<sup>st</sup> generic analyzer of multithreaded program
- Implements the Java bytecode semantics
- Generic w.r.t.
  - > Numerical domain
    - Interval, sign, parity, congruence
  - > Memory model
    - Happens-before one
  - > Property of interest
    - Multithreading: data race, deadlock, determinism
    - Well-known: division by zero, access to null, etc..

<http://www.pietro.ferrara.name/checkmate>

# Structure



# Experimental results

- Applied to
  - > A set of examples taken from [JMM]
    - Precise
  - > An incremental application
    - Complexity linear or at most quadratic
  - > Some benchmarks taken from [ORD, BENCH]
    - Programs with
      - > An unbounded number of threads
      - > Some thousands of statements

[JMM] J. Manson, W. Pugh, and S. V. Adve. *The Java memory model*. In ACM Press, editor, Proceedings of POPL '05, 2005.

[ORD] 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

<b>Program</b>	<b>St.</b>	<b>Th.</b>	<b>Top</b>	<b>Sign</b>	<b>Int.</b>	<b>Par.</b>	<b>Cong.</b>
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''

# Outline

1. Introduction
2. ~~Happens-before memory model~~
  - ~~Definition in fixpoint form and abstraction~~
3. ~~Determinism of multithreaded programs~~
  - ~~New property~~
4. ~~Domain and semantics of Java bytecode~~
  - ~~Low-level domain, specific alias analysis~~
5. ~~Checkmate~~
  - ~~1<sup>st</sup> generic 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
- .Net: safe environment of execution
  - > Exception: unsafe code
  - > **Direct access** to the memory
- No guarantees on direct memory accesses
  - > **Buffer overrun**

# Background

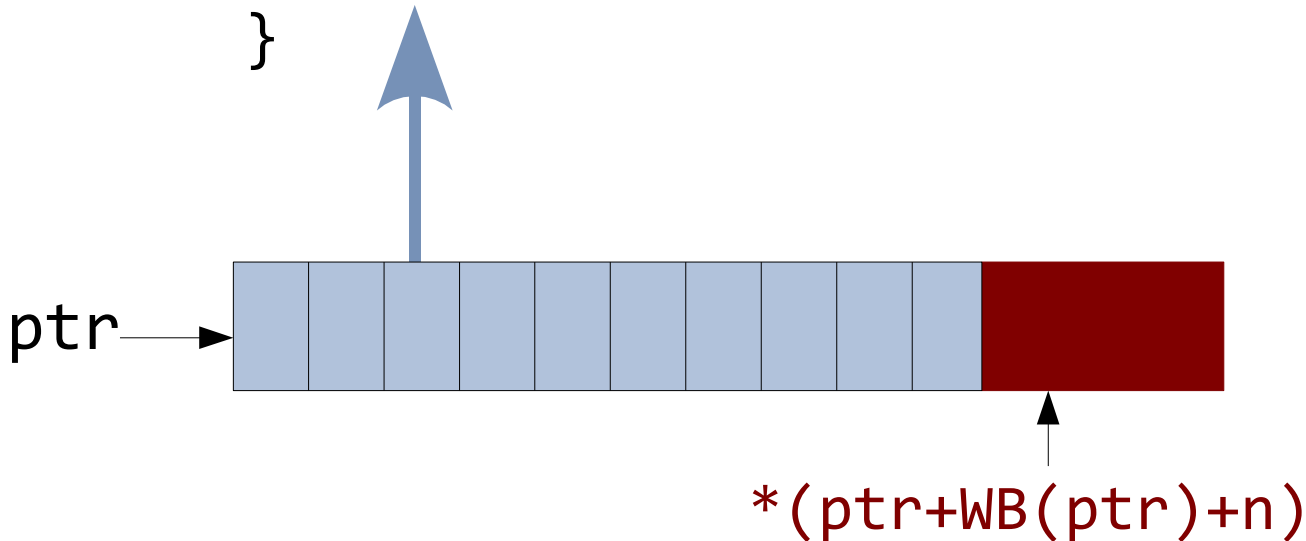
- Goal: apply Clousot to unsafe code
- Our solution:
- Combination with **contracts**
  - > Method boundary annotation
    - Automatically infer loop invariants
  - > Lightweight domains
    - Stripes: new relational domain
    - Combine it with well-known numerical domains
  - > **Scalability**
    - .Net libraries analyzed in a couple of minutes

# Example: InitToZero

```
unsafe void InitToZero(int* ptr, uint len)
{
    for (int i = 0; i < len; i++)
        *(ptr + i) = 0;
}
```

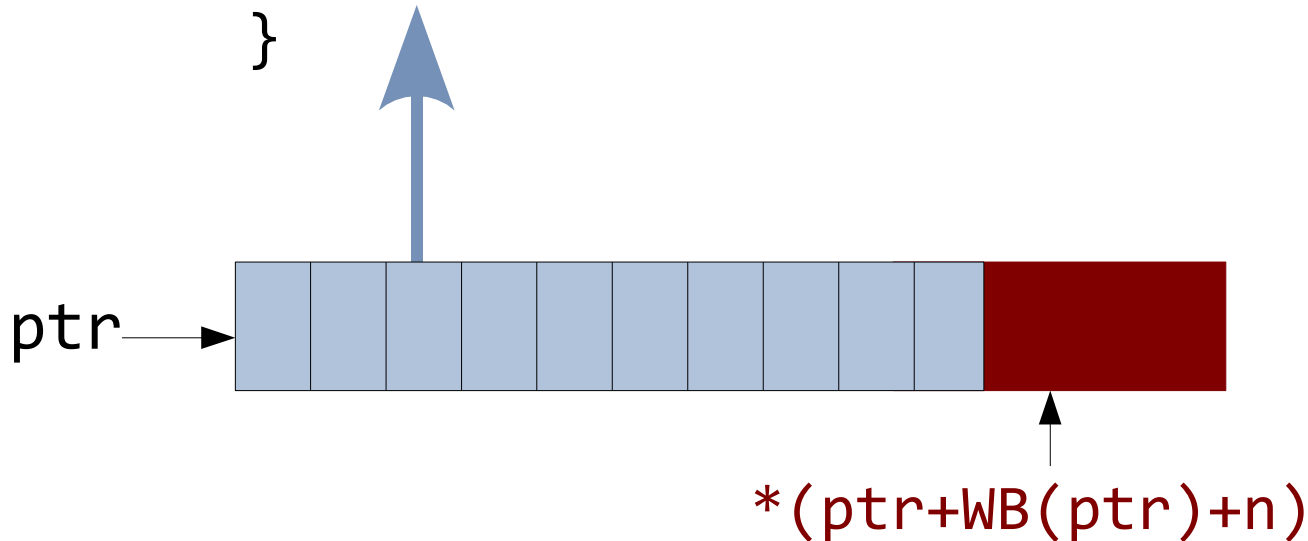
# Example: InitToZero

```
unsafe void InitToZero(int* ptr, uint len)
{
  for (int i = 0; i < len; i++)
    *(ptr + i) = 0;
}
```



# Example: InitToZero

```
unsafe void InitToZero(int* ptr, uint len)
{
  for (int i = 0; i < len; i++)
    *(ptr + i) = 0;
}
```



Precondition needed:

*"at least  $len * \text{sizeof}(int)$  bytes allocated starting from  $ptr$ "*

# 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;
}
```

Infer: loop invariant  $0 \leq i < \text{len}$

Specify precondition

# Example: InitToZero

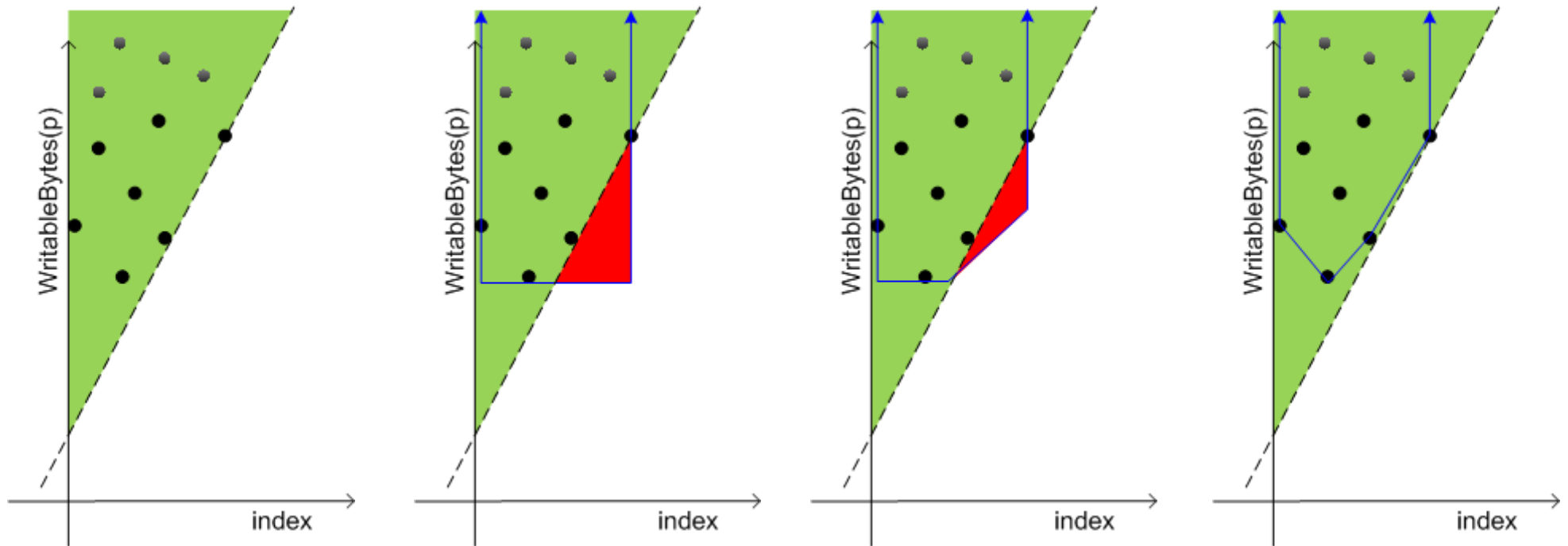
```
unsafe void InitToZero(int* ptr, uint len)
{
    Contract.Requires(Contract.WB(ptr) ≥ len*4);
    for (int i = 0; i < len; i++)
        *(ptr + i) = 0;
}
```

Infer: loop invariant  $0 \leq i < \text{len}$

Specify precondition

Prove no buffer overrun

# Numerical Domains



Concrete values

Intervals

Octagons

Polyhedra

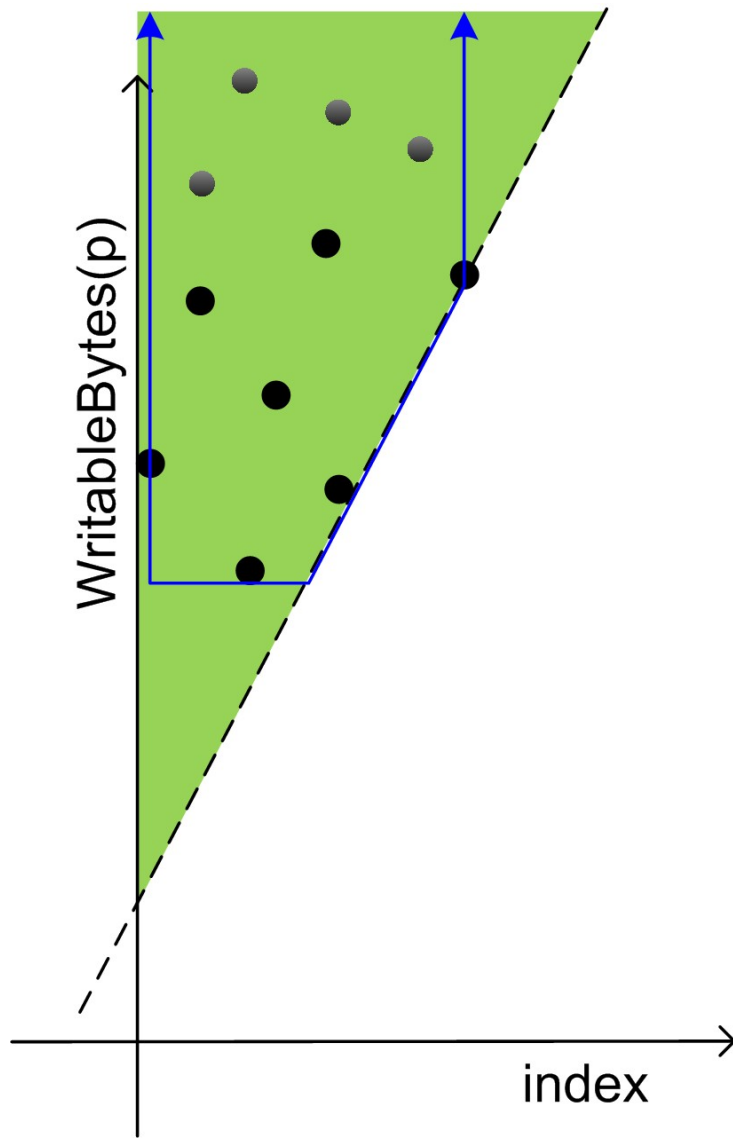
$$O(n)$$

$$O(n^3)$$

$$O(2^n)$$

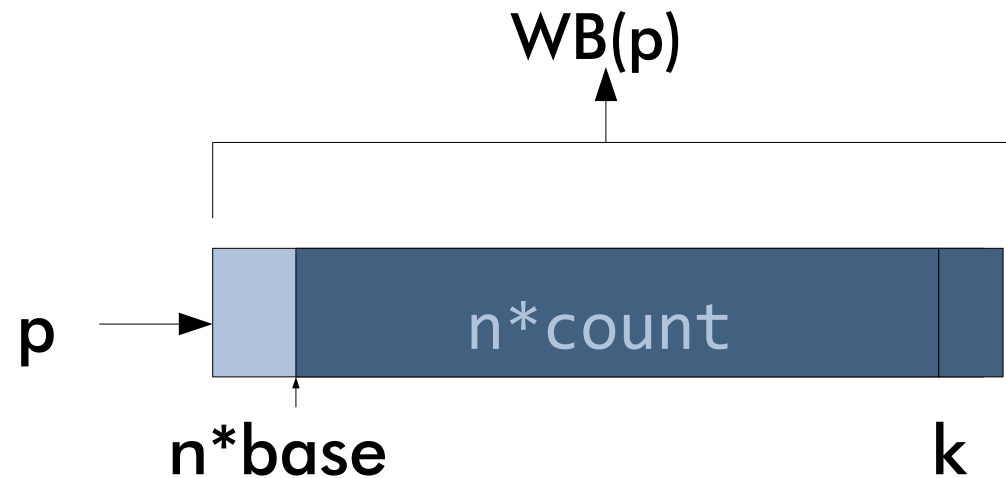
- **Polyhedra: only monolithic domain precise enough**
  - > **Exponential** complexity in practice

# Numerical Domains - Stripes



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

- **Relational**
- **Precise**



- **Linear complexity in practice**

# Example: InitToZero

$\text{check}(\text{WB}(p) \geq \text{sizeof}(x) + \text{exp} * \text{sizeof}(*p), \bar{s}) = \text{true}$

$\text{check}(\text{exp} \geq 0, \bar{s}) = \text{true}$

---

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

```
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

$\text{check}(\text{WB}(p) \geq \text{sizeof}(x) + \text{exp} * \text{sizeof}(*p), \bar{s}) = \text{true}$

$\text{check}(\text{exp} \geq 0, \bar{s}) = \text{true}$

---

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

```
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:**

$\text{WB}(ptr) \geq 4 * \text{len}$

$\text{len} \geq i + 1$

$\text{WB}(ptr) \geq 4 * i + 4$

# Example: InitToZero

check( $\text{WB}(p) \geq \text{sizeof}(x) + \text{exp} * \text{sizeof}(*p)$ ,  $\bar{s}$ ) = true  
check( $\text{exp} \geq 0$ ,  $\bar{s}$ ) = true

---

$$\mathbb{F} \llbracket *(p + \text{exp}) = x \rrbracket (\bar{s}) \rightarrow \bar{s}$$

```
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:**  
 $\text{WB}(ptr) \geq 4 + i * 4$

**Infer:**  
 $\text{WB}(ptr) \geq 4 * len$   
 $len \geq i + 1$   
 $\text{WB}(ptr) \geq 4 * i + 4$

# Example: InitToZero

check( $WB(p) \geq \text{sizeof}(x) + \text{exp} * \text{sizeof}(*p)$ ,  $\bar{s}$ ) = true  
check( $\text{exp} \geq 0$ ,  $\bar{s}$ ) = true

$$\mathbb{F} \llbracket *(p + \text{exp}) = x \rrbracket (\bar{s}) \rightarrow \bar{s}$$

```
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:**  
 $WB(ptr) \geq 4 + i * 4$

**Validate**

**Infer:**  
 $WB(ptr) \geq 4 * len$   
 $len \geq i + 1$   
 $WB(ptr) \geq 4 * i + 4$

# Intervals

check( $WB(p) \geq \text{sizeof}(x) + \text{exp} * \text{sizeof}(*p)$ ,  $\bar{s}$ ) = true

check( $\text{exp} \geq 0$ ,  $\bar{s}$ ) = true


---

$\mathbb{F} \llbracket *(p + \text{exp}) = x \rrbracket (\bar{s}) \rightarrow \bar{s}$

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

```
    *(ptr + i) = 0;
```

$WB(ptr) \geq 4 * len$   
 $len \geq i + 1$   
 $WB(ptr) \geq 4 * i + 4$



- **Stripes does not prove  $i \geq 0$**

# Intervals

check( $WB(p) \geq \text{sizeof}(x) + \text{exp} * \text{sizeof}(*p), \bar{s}) = \text{true}$   
check( $\text{exp} \geq 0, \bar{s}) = \text{true}$

---

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

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

$WB(ptr) \geq 4 * len$   
 $len \geq i + 1$   
 $WB(ptr) \geq 4 * i + 4$

- Stripes does not prove  $i \geq 0$

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

$i = [0..+\infty]$

- 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

# Outline

1. Introduction
2. ~~Happens-before memory model~~
  - ~~Definition in fixpoint form and abstraction~~
3. ~~Determinism of multithreaded programs~~
  - ~~New property~~
4. ~~Domain and semantics of Java bytecode~~
  - ~~Low-level domain, specific alias analysis~~
5. ~~Checkmate~~
  - ~~1<sup>st</sup> generic analyzer of multithreaded programs~~
6. ~~Static analysis of unsafe code~~
  - ~~An industrial application of generic analyzer~~

# Conclusion

- Definition of HBMM in a **fixpoint form**
- **Abstracted** with a computable semantics
- 1<sup>st</sup> generic static **analysis** sound w.r.t. MM

**[Ferr08]** P. Ferrara, "*Static analysis via abstract interpretation of the happens-before memory model*", in Springer editor, Proceedings of the 2nd International Conference on Tests and Proofs (TAP 2008), volume 4966 of Lecture Notes in Computer Science, Prato, Italy, April 9-11, 2008

# Conclusion

- **Determinism:** **new** property
  - > Focused on **communications** via shared memory
- On concrete and abstract domains and semantics
- **Relaxed** in three different ways
  - > **Flexible**

**[Ferr08b]** P. Ferrara, "*Static analysis of the determinism of multithreaded programs*", in IEEE Computer Society, editor, Proceedings of the Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), Cape Town, South Africa, November 10-14, 2008

# Conclusion

- **Low-level** domain and semantics
  - > Java bytecode
  - > Core: alias analysis
- Initially applied to data race detection
- Then extended to build up a **generic analyzer**

**[Ferr08a]** P. Ferrara, "*A fast and precise analysis for data race detection*", in Elsevier editor, Proceedings of the Third Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'08), Electronic Notes in Theoretical Computer Science, Budapest, Hungary, April 6, 2008

# Conclusion

- 1<sup>st</sup> generic analyzer of multithreaded programs
  - > Property (determinism and weak determinism)
  - > Numerical domain
  - > Memory model (HBMM)
- Applied to
  - > Case studies taken from the Java MM
  - > Incremental application
  - > Benchmarks
- Experimental results **encouraging**

P. Ferrara, "*Checkmate: a generic static analyzer of Java multithreaded programs*", to be submitted

# Conclusion

- Static analysis of buffer overrun
- Implemented in Clousot
  - > Generic industrial static analyzer of MSIL
- Precise and scalable

**[FLF08]** P. Ferrara, F. Logozzo and M. Fähndrich, "*Safer unsafe code for .NET*", in ACM Press, editor, Proceedings of the 23rd ACM Conference on Object-oriented Programming (OOPSLA 2008), Nashville, USA, October 19-23, 2008

# Future work

- How to refine the **MM** and its analysis
  - > Other synchronizations have to be considered
  - > Interesting restrictions of the **JMM**
- How to **project** the deterministic property
  - > E.g. synchronized accesses
- **Refine** bytecode domain and semantics
  - > Goal: apply to numerical **relational domains**
- **Implement** it in Checkmate

# Question time

# Thank you!

# Future work

- Whole program analysis
  - > Limit: do 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