Static Analysis via Abstract Interpretation of the Happens-Before Memory Model

Pietro Ferrara

Ecole Polytechnique, Paris
F-91128 Palaiseau (France)
Università Ca' Foscari di Venezia
I-30170 Venezia (Italy)
Pietro.Ferrara@polytechnique.edu
ferrara@dsi.unive.it

June 20, 2008

Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

Multithreading

- Multicore architectures
  - many parallel tasks
- More difficult than sequential programs [SUT05]
  - interleaving of the executions of different threads
  - implicit communications through the shared memory
  - specification of the memory model [LEE06]

Sequential programs

- Trace semantics - Execution represented as a trace of states:

  $1: \texttt{for(int i} = 0;\ \texttt{i} < \texttt{arr.length};\ \texttt{i}++)$

  $2: \quad \texttt{arr[i]} = 0;$

- Represented by $\sigma_0 \xrightarrow{1} \sigma_1 \xrightarrow{2} \sigma_2 \xrightarrow{1} \ldots$

Sequential programs

- Trace semantics - Execution represented as a trace of states:

  $1: \ \texttt{for}(\texttt{int i} = 0; \ \texttt{i} < \texttt{arr.length}; \ \texttt{i}++)$

  $2: \quad \texttt{arr}[\texttt{i}] = 0;$
- Represented by $\sigma_0 \xrightarrow{1} \sigma_1 \xrightarrow{2} \sigma_2 \xrightarrow{1} \ldots$
- We know exactly the sequence of statements executed and the order in which they are executed

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- If executed on a single core architecture:

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- If executed on a single core architecture:

$$\sigma_0 \xrightarrow{S1:System} \sigma_1$$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- If executed on a single core architecture:

$$\sigma_0 \overset{S1:System}{\longrightarrow} \sigma_1 \overset{S2:System}{\longrightarrow} \sigma_2$$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- If executed on a single core architecture:

$$\sigma_0 \xrightarrow{S1:System} \sigma_1 \xrightarrow{S2:System} \sigma_2 \xrightarrow{T1:MyThread} \sigma_3$$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum += a[j];
	}

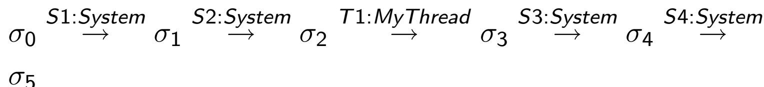
- If executed on a single core architecture:

$$\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1 \stackrel{S2:System}{\longrightarrow} \sigma_2 \stackrel{T1:MyThread}{\longrightarrow} \sigma_3 \stackrel{S3:System}{\longrightarrow} \sigma_4$$

Multithreaded programs

System	MyThread
`S1 : t = new MyThread(a);`	`public void run(){`
`S2 : t.start();`	`T1 : int sum = 0;`
`S3 : for(int i = 0; i < a.length; i++)`	`T2 : for(int j = 0; j < a.length; j++)`
`S4 : a[i] = 0;`	`T3 : sum += a[j];`
	`}`

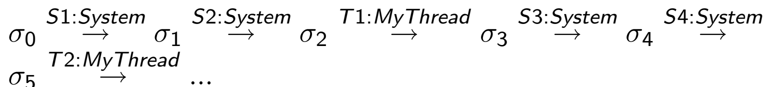
- If executed on a single core architecture:



Multithreaded programs

System	MyThread
S1 : t = new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum = 0;
S3 : for(int i = 0; i < a.length; i++)	T2 : for(int j = 0; j < a.length; j++)
S4 : a[i] = 0;	T3 : sum+ = a[j];
	}

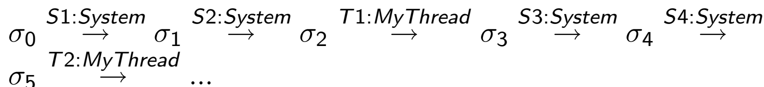
- If executed on a single core architecture:



Multithreaded programs

System	MyThread
S1 : t = new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum = 0;
S3 : for(int i = 0; i < a.length; i ++)	T2 : for(int j = 0; j < a.length; j ++)
S4 : a[i] = 0;	T3 : sum+ = a[j];
	}

- If executed on a single core architecture:



$$\sigma_0 \xrightarrow{S1:System} \sigma_1 \xrightarrow{S2:System} \sigma_2 \xrightarrow{T1:MyThread} \sigma_3 \xrightarrow{S3:System} \sigma_4 \xrightarrow{S4:System} \sigma_5 \xrightarrow{T2:MyThread} \ldots$$

- Casual interleaving between the executions of different threads

Multithreaded programs

System	MyThread

S1 : `t = new MyThread(a);`
S2 : `t.start();`
S3 : `for(int i = 0; i < a.length; i++)`
S4 : `a[i] = 0;`
	`public void run(){`
	T1 : `int sum = 0;`
	T2 : `for(int j = 0; j < a.length; j++)`
	T3 : `sum += a[j];`
	`}`

- Dual-core architecture:

Core 1: σ_0

Core 2: σ_0

Multithreaded programs

System	MyThread
S1 : t = new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum = 0;
S3 : for(int i = 0; i < a.length; i++)	T2 : for(int j = 0; j < a.length; j++)
S4 : a[i] = 0;	T3 : sum+ = a[j];
	}

- Dual-core architecture:

Core 1: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1$

Core 2: σ_0

Multithreaded programs

System	MyThread

S1 : `t = new MyThread(a);`
S2 : `t.start();`
S3 : `for(int i = 0; i < a.length; i++)`
S4 : `a[i] = 0;`
	`public void run(){`
	T1 : `int sum = 0;`
	T2 : `for(int j = 0; j < a.length; j++)`
	T3 : `sum += a[j];`
	`}`

- Dual-core architecture:

Core 1: $\sigma_0 \overset{S1:System}{\longrightarrow} \sigma_1 \overset{S2:System}{\longrightarrow} \sigma_2$

Core 2: σ_0

Multithreaded programs

System	MyThread
$\texttt{S1}: \ \texttt{t} = \texttt{new MyThread(a);}$	$\texttt{public void run()}\{$
$\texttt{S2}: \ \texttt{t.start();}$	$\texttt{T1}: \ \texttt{int sum} = 0;$
$\texttt{S3}: \ \texttt{for(int i} = 0; \texttt{i} < \texttt{a.length}; \texttt{i}++)$	$\texttt{T2}: \ \texttt{for(int j} = 0; \texttt{j} < \texttt{a.length}; \texttt{j}++)$
$\texttt{S4}: \ \ \texttt{a[i]} = 0;$	$\texttt{T3}: \ \ \texttt{sum}+= \texttt{a[j]};$
	$\}$

- Dual-core architecture:

Core 1: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1 \stackrel{S2:System}{\longrightarrow} \sigma_2$

Core 2: $\sigma_0 \stackrel{T1:MyThread}{\longrightarrow} \sigma_1'$

Multithreaded programs

System	MyThread
S1 : t = new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum = 0;
S3 : for(int i = 0; i < a.length; i++)	T2 : for(int j = 0; j < a.length; j++)
S4 : a[i] = 0;	T3 : sum+ = a[j];
	}

- Dual-core architecture:

Core 1: $\sigma_0 \xrightarrow{S1:System} \sigma_1 \xrightarrow{S2:System} \sigma_2 \xrightarrow{S3:System} \sigma_3$

Core 2: $\sigma_0 \xrightarrow{T1:MyThread} \sigma'_1 \xrightarrow{T2:MyThread} \sigma'_2$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 :   a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- Dual-core architecture:

Core 1: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1 \stackrel{S2:System}{\longrightarrow} \sigma_2 \stackrel{S3:System}{\longrightarrow} \sigma_3 \stackrel{S4:System}{\longrightarrow} \ldots$

Core 2: $\sigma_0 \stackrel{T1:MyThread}{\longrightarrow} \sigma'_1 \stackrel{T2:MyThread}{\longrightarrow} \sigma'_2$

Multithreaded programs

System	MyThread

S1 : $t = \texttt{new MyThread}(a);$
S2 : $t.\texttt{start}();$
S3 : $\texttt{for}(\texttt{int } i = 0; i < a.\texttt{length}; i{+}{+})$
S4 : $a[i] = 0;$
	public void run(){
	T1 : $\texttt{int } sum = 0;$
	T2 : $\texttt{for}(\texttt{int } j = 0; j < a.\texttt{length}; j{+}{+})$
	T3 : $sum += a[j];$
	}

- Dual-core architecture:

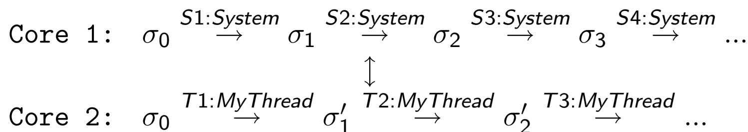
Core 1: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1 \stackrel{S2:System}{\longrightarrow} \sigma_2 \stackrel{S3:System}{\longrightarrow} \sigma_3 \stackrel{S4:System}{\longrightarrow} \ldots$

Core 2: $\sigma_0 \stackrel{T1:MyThread}{\longrightarrow} \sigma'_1 \stackrel{T2:MyThread}{\longrightarrow} \sigma'_2 \stackrel{T3:MyThread}{\longrightarrow} \ldots$

Multithreaded programs

System	MyThread
S1 : t = new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum = 0;
S3 : for(int i = 0; i < a.length; i ++)	T2 : for(int j = 0; j < a.length; j ++)
S4 : a[i] = 0;	T3 : sum+ = a[j];
	}

- Dual-core architecture:



Multithreaded programs

System	MyThread
S1 : $t = $ new MyThread(a);	public void run(){
S2 : t.start();	T1 : int sum $= 0$;
S3 : for(int i $= 0$; i $<$ a.length; i $++$)	T2 : for(int j $= 0$; j $<$ a.length; j $++$)
S4 : a[i] $= 0$;	T3 : sum $+= $ a[j];
	}

- Dual-core architecture:

Core 1: σ_0

Core 2: σ_0

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- Dual-core architecture:

Core 1: σ_0

Core 2: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma'_1$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i ++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j ++)
	T3 : sum+ = a[j];
	}

- Dual-core architecture:

Core 1: $\sigma_0 \stackrel{S2:System}{\longrightarrow} \sigma_2$

Core 2: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1'$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i ++)
S4 :   a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j ++)
	T3 : sum+ = a[j];
	}

- Dual-core architecture:

Core 1: $\sigma_0 \stackrel{S2:System}{\longrightarrow} \sigma_2 \stackrel{T1:MyThread}{\longrightarrow} \sigma_3$

Core 2: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1'$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum += a[j];
	}

- Dual-core architecture:

Core 1: $\sigma_0 \stackrel{S2:System}{\longrightarrow} \sigma_2 \stackrel{T1:MyThread}{\longrightarrow} \sigma_3 \stackrel{T2:MyThread}{\longrightarrow} \sigma_4$

Core 2: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1' \stackrel{S3:System}{\longrightarrow} \sigma_2'$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- Dual-core architecture:

$$\texttt{Core 1:} \quad \sigma_0 \overset{S2:System}{\longrightarrow} \sigma_2 \overset{T1:MyThread}{\longrightarrow} \sigma_3 \overset{T2:MyThread}{\longrightarrow} \sigma_4$$

$$\texttt{Core 2:} \quad \sigma_0 \overset{S1:System}{\longrightarrow} \sigma'_1 \overset{S3:System}{\longrightarrow} \sigma'_2 \overset{S4:System}{\longrightarrow} \dots$$

Multithreaded programs

System	MyThread
S1 : $\texttt{t} = \texttt{new MyThread(a);}$ S2 : $\texttt{t.start();}$ S3 : $\texttt{for(int i} = 0; \texttt{i} < \texttt{a.length}; \texttt{i}++)$ S4 : $\texttt{a[i]} = 0;$	$\texttt{public void run()}\{$ T1 : $\texttt{int sum} = 0;$ T2 : $\texttt{for(int j} = 0; \texttt{j} < \texttt{a.length}; \texttt{j}++)$ T3 : $\texttt{sum} += \texttt{a[j]};$ $\}$

- Dual-core architecture:

Core 1: $\sigma_0 \stackrel{S2:System}{\longrightarrow} \sigma_2 \stackrel{T1:MyThread}{\longrightarrow} \sigma_3 \stackrel{T2:MyThread}{\longrightarrow} \sigma_4 \stackrel{T3:MyThread}{\longrightarrow} \ldots$

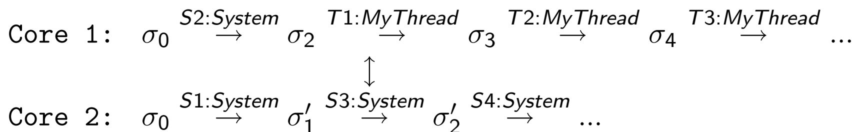
Core 2: $\sigma_0 \stackrel{S1:System}{\longrightarrow} \sigma_1' \stackrel{S3:System}{\longrightarrow} \sigma_2' \stackrel{S4:System}{\longrightarrow} \ldots$

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
 S2 : t.start();
 S3 : for(int i = 0; i < a.length; i + +)
 S4 : a[i] = 0;
	public void run(){
	 T1 : int sum = 0;
	 T2 : for(int j = 0; j < a.length; j + +)
	 T3 : sum+ = a[j];
	 }

- Dual-core architecture:



Memory model

- Many architectures, many compilers, ...
- What is legal and what is not?

Memory model

- Many architectures, many compilers, ...
- What is legal and what is not?
- The memory model defines which multithreaded executions are legal and which are not!

Memory model

- Many architectures, many compilers, ...
- What is legal and what is not?
- The memory model defines which multithreaded executions are legal and which are not!
  - Memory Model $\supseteq$ executions on single core architectures
  - Memory Model $\supseteq$ executions on dual core architectures
  - ...

Memory model

- Many architectures, many compilers, ...
- What is legal and what is not?
- The memory model defines which multithreaded executions are legal and which are not!
  - Memory Model $\supseteq$ executions on single core architectures
  - Memory Model $\supseteq$ executions on dual core architectures
  - ...
- It does not define HOW a multithreaded program is executed
- It defines WHAT a multithreaded program is allowed to see
  - i.e.: which values can be read from the shared memory

Memory Model

- Two opposite needs:
  - restrict the non-deterministic behaviors
  - allow as many compiler optimizations as possible
- e.g. sequential consistency:
  - "...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [LAM79]
- too much restrictive in practice, it does not allow the most part of optimizations

An example

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

- Supposing that at the beginning $i = 0, j = 0$

An example

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

- Supposing that at the beginning $i = 0, j = 0$
- May the exception be thrown?

An example

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

- Supposing that at the beginning $i = 0, j = 0$
- May the exception be thrown?
  - sequential consistency: NO!

An example

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

- Supposing that at the beginning $i = 0, j = 0$
- May the exception be thrown?
  - sequential consistency: NO!
  - practical needs: YES!

An example

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

- Supposing that at the beginning $i = 0, j = 0$
- May the exception be thrown?
  - sequential consistency: NO!
  - practical needs: YES!
    - swap of independent statements

The Happens-Before Memory Model

- Program order:
  - intra-thread order of statements
- Synchronizes-with relation: involve two **synchronized** actions. E.g. the acquisition and the release of a monitor
- Happens-before order [LAM78]: a_1 happens-before a_2 if
  - a_1 appears before a_2 in the program order
  - a_2 synchronizes-with a_1
  - you can reach a_2 by following happens-before edges starting from a_1

Reads

- Consistency rule:
  - a read r of a variable v is allowed to see a write w to v if:
    - r does not happens-before w
    - there is not any w' to v that happens-before r and such that w happens-before it

Reads

- Consistency rule:
  - a read r of a variable v is allowed to see a write w to v if:
    - r does not happens-before w
    - there is not any w' to v that happens-before r and such that w happens-before it
- The happens-before memory model is an approximation of the Java one
  - If a behavior is allowed by the Java MM, it is allowed also by the happens-before MM

Synchronizes-with

- Many different ways of synchronization:
  - mutual exclusion on monitors
  - `wait` and `notify` on objects
  - the first action of a thread synchronizes-with the statement that launched it
  - ...

Synchronizes-with

- Many different ways of synchronization:
  - mutual exclusion on monitors
  - `wait` and `notify` on objects
  - the first action of a thread synchronizes-with the statement that launched it
  - ...
- We are generic w.r.t. the programming language
- We focus on
  - mutual exclusion
  - launch of a thread

Synchronizes-with

- Many different ways of synchronization:
  - mutual exclusion on monitors
  - `wait` and `notify` on objects
  - the first action of a thread synchronizes-with the statement that launched it
  - ...
- We are generic w.r.t. the programming language
- We focus on
  - mutual exclusion
  - launch of a thread
- Other synchronizations... future work!

Our contribution

- A **static analysis**
- On all the multithreaded programs
- Sound w.r.t. the happens-before memory model
- Based on the **abstract interpretation** theory

Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

Single-thread Domain

- Generic w.r.t. the programming language ...

Single-thread Domain

- Generic w.r.t. the programming language ...
- ... we do not formally define in the details the concrete domain!

Single-thread Domain

- Generic w.r.t. the programming language ...
- ... we do not formally define in the details the concrete domain!
- Some functions provide us the essential information
  - $\mathit{shared} : \mathit{St} \mapsto \mathit{Sh}$
  - $\mathit{action} : \mathit{St} \mapsto \bot_a \cup (\{\mathtt{r}, \mathtt{w}\} \times \mathit{Loc} \times \mathit{Val})$
  - $\mathit{synchronised} : \mathit{St} \mapsto \wp(\mathit{Sync})$
- Parameterized also on the intra-thread transfer function $\xrightarrow{\circ}$

Multithread Domain

- For each thread we collect the trace representing its execution

$$\Psi : \mathit{TId} \longrightarrow \mathit{St}^{\vec{+}}$$

Multithread Domain

- For each thread we collect the trace representing its execution

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

$$\Omega : \mathit{TId} \rightarrow ((\mathit{TId} \times \mathit{Integer}) \cup \bot_{\Omega})$$

- $t_1 \mapsto (t_2, 5)$ means that thread t_1 has been launched by thread t_2 at the instruction at program counter 5

Multithread Domain

- For each thread we collect the trace representing its execution

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

$$\Omega : TId \rightarrow ((TId \times Integer) \cup \bot_{\Omega})$$

- $t_1 \mapsto (t_2, 5)$ means that thread t_1 has been launched by thread t_2 at the instruction at program counter 5
- We abstract away the inter-thread order of execution!

Multithreaded programs

System	MyThread

S1 : t = new MyThread(a);
S2 : t.start();
S3 : for(int i = 0; i < a.length; i++)
S4 : a[i] = 0;
	public void run(){
	T1 : int sum = 0;
	T2 : for(int j = 0; j < a.length; j++)
	T3 : sum+ = a[j];
	}

- Ψ :

$$\texttt{System} \mapsto \{\sigma_0 \xrightarrow{S1} \sigma_1 \xrightarrow{S2} \sigma_2 \xrightarrow{S3} \sigma_3 \xrightarrow{S4} \sigma_4 \xrightarrow{S3} \sigma_5 \xrightarrow{S4} \cdots\}$$

$$\texttt{MyThread} \mapsto \{\sigma'_0 \xrightarrow{T1} \sigma'_1 \xrightarrow{T2} \sigma'_2 \xrightarrow{T3} \sigma'_3 \xrightarrow{T2} \sigma'_4 \xrightarrow{T3} \sigma'_5 \xrightarrow{T2} \cdots\}$$

Multithreaded programs

System	MyThread
$\texttt{S1}: \ \texttt{t} = \texttt{new MyThread}(\texttt{a});$ $\texttt{S2}: \ \texttt{t.start}();$ $\texttt{S3}: \ \texttt{for}(\texttt{int i} = 0; \texttt{i} < \texttt{a.length}; \texttt{i}++)$ $\texttt{S4}: \quad \texttt{a}[\texttt{i}] = 0;$	$\texttt{public void run}()\{$ $\texttt{T1}: \ \texttt{int sum} = 0;$ $\texttt{T2}: \ \texttt{for}(\texttt{int j} = 0; \texttt{j} < \texttt{a.length}; \texttt{j}++)$ $\texttt{T3}: \quad \texttt{sum} += \texttt{a}[\texttt{j}];$ $\}$

- Ψ :

$$\begin{aligned} \texttt{System} &\mapsto \{\sigma_0 \xrightarrow{S1} \sigma_1 \xrightarrow{S2} \sigma_2 \xrightarrow{S3} \sigma_3 \xrightarrow{S4} \sigma_4 \xrightarrow{S3} \sigma_5 \xrightarrow{S4} \cdots\} \\ \texttt{MyThread} &\mapsto \{\sigma'_0 \xrightarrow{T1} \sigma'_1 \xrightarrow{T2} \sigma'_2 \xrightarrow{T3} \sigma'_3 \xrightarrow{T2} \sigma'_4 \xrightarrow{T3} \sigma'_5 \xrightarrow{T2} \cdots\} \end{aligned}$$

- $\Omega : \{\texttt{MyThread} \mapsto (\texttt{System}, \texttt{S2}), \texttt{System} \mapsto \bot_\Omega\}$

Intra-thread semantics

- *visible* function: return all the values written in parallel by other threads that may be seen following the HB memory model
- *step* function: defines a single step in the computation. Read from the shared memory one of the visible values $\rightarrow$ nondeterministic choices

Definition ($\mathbb{S}^\circ$)

$$\begin{aligned}
\mathbb{S}^\circ \quad & : \Psi \times \Omega \times \mathit{TId} \mapsto \wp(\vec{St^+}) \\
\mathbb{S}^\circ \quad & [\![f, r, t]\!] = \mathit{lfp}_{\emptyset}^{\subseteq} \lambda T.\{\sigma_0\} \cup \{\sigma_0 \rightarrow \cdots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \\
& \qquad \sigma_0 \rightarrow \cdots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \mathit{step}(t, f, r)\}
\end{aligned}$$

Definition (*step* function)

$$
\begin{array}{ll}
\mathit{step} & : \mathit{TId} \times \Psi \times \Omega \mapsto \wp(\mathit{St}) \\
\mathit{step} & (t, f, s) = \{\sigma\} \text{ where } f(t) = \sigma_0 \rightarrow \cdots \rightarrow \sigma_i \text{ and} \\
(1) & \sigma_i \overset{\circ}{\rightarrow} \sigma \quad \text{ if } \pi_1(\mathit{action}(\sigma_i)) \neq \mathtt{r} \\
\\
(2a) & \sigma_i \overset{\circ}{\rightarrow} \sigma \vee \quad \text{ if } \pi_1(\mathit{action}(\sigma_i)) = \mathtt{r} \\
(2b) & \exists v \in \mathit{visible}(t, \pi_2(\mathit{action}(\sigma_i)), \mathit{synchronised}(\sigma_i), f, s(t)) : \\
 & l = \pi_2(\mathit{action}(\sigma_i)) \\
 & \sigma' = \mathit{set_shared}(\sigma_i, \mathit{shared}(\sigma_i)[l \mapsto v]), \sigma' \overset{\circ}{\rightarrow} \sigma
\end{array}
$$

Multithread semantics

- Each time we compute the single thread semantics of a thread it may expose new values to other threads
- Iterates the single-thread semantics until a fixpoint is reached

Definition ($\mathbb{S}^{\parallel}$)

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

An example

System	Thread 1	Thread 2
new Thread1().start(); new Thread2().start();	i = 1; j = 1;	if(j == 1&&i == 0) throw new Exception();

An example - 1st iteration

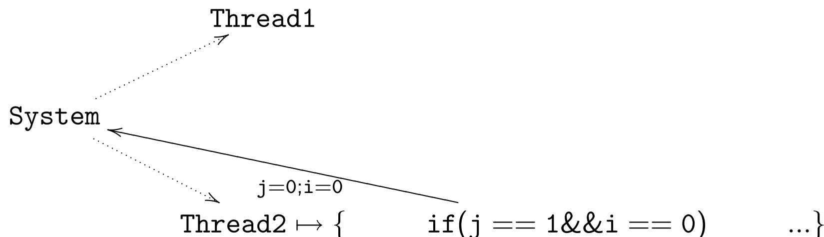
System	Thread 1	Thread 2
`new Thread1().start();` `new Thread2().start();`	`i = 1;` `j = 1;`	`if(j == 1&&i == 0)` `throw new Exception();`

`System`

- The previous state is empty
- Only an active thread: the one launched by the system!
- We compute only the semantics of that thread

An example - 2nd iteration

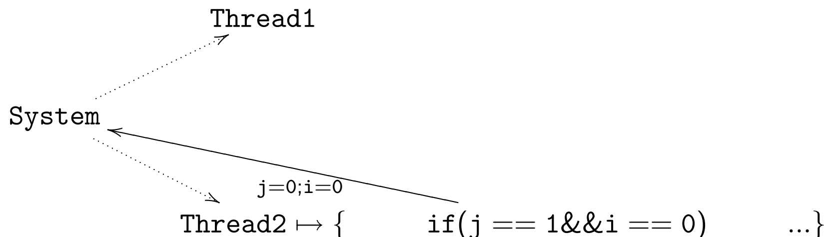
System	Thread 1	Thread 2
`new Thread1().start();`	`i = 1;`	`if(j == 1&&i == 0)`
`new Thread2().start();`	`j = 1;`	`throw new Exception();`



- We compute the semantics of three threads
- ... but the previous state exposes only the values written by `System`!

An example - 2nd iteration

System	Thread 1	Thread 2
`new Thread1().start();` `new Thread2().start();`	$i = 1;$ $j = 1;$	$\texttt{if}(j == 1 \&\& i == 0)$ `throw new Exception();`

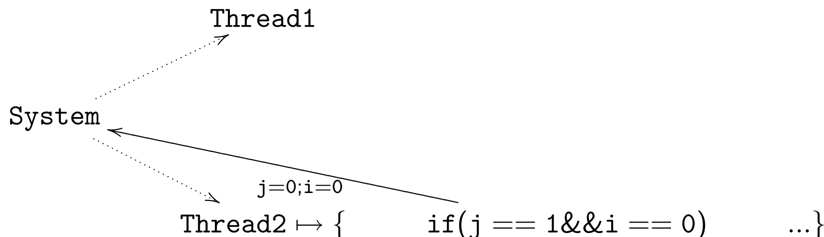


- We compute the semantics of three threads
- ... but the previous state exposes only the values written by `System`!

$\text{visible}(\texttt{j}) = \{0\}$

An example - 2nd iteration

System	Thread 1	Thread 2
`new Thread1().start();` `new Thread2().start();`	`i = 1;` `j = 1;`	`if(j == 1&&i == 0)` `throw new Exception();`

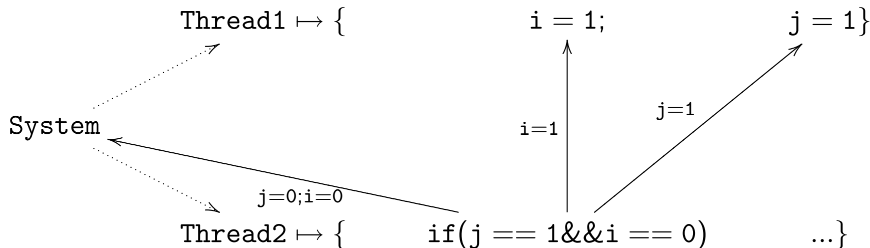


- We compute the semantics of three threads
- ... but the previous state exposes only the values written by `System`!

visible(`j`) $= \{0\}$
visible(`i`) $= \{0\}$

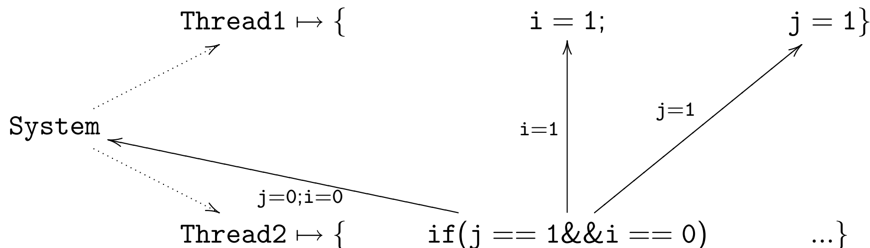
An example - 3rd iteration

System	Thread 1	Thread 2
new Thread1().start(); new Thread2().start();	i = 1; j = 1;	if(j == 1&&i == 0) throw new Exception();



An example - 3rd iteration

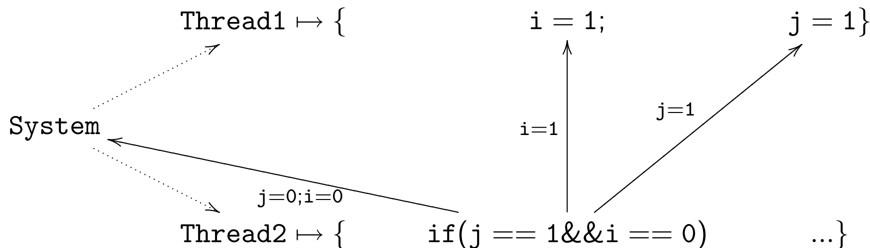
System	Thread 1	Thread 2
`new Thread1().start();` `new Thread2().start();`	`i = 1;` `j = 1;`	`if(j == 1&&i == 0)` `throw new Exception();`



$\text{visible}(\texttt{j}) = \{0, 1\}$

An example - 3rd iteration

System	Thread 1	Thread 2
`new Thread1().start();` `new Thread2().start();`	`i = 1;` `j = 1;`	`if(j == 1&&i == 0)` `throw new Exception();`



visible(j) = {0, 1}
visible(i) = {0, 1}

Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

Domain

- Exactly the same components of the concrete domain, but dealing with abstract elements
- Functions:
  - $shared^{\#} : \overline{St} \mapsto \overline{Sh}$
  - $action^{\#} : \overline{St} \mapsto \overline{\bot}_a \cup (\{\mathtt{r}, \mathtt{w}\} \times \overline{Loc} \times \overline{Val})$
  - $synchronised^{\#} : \overline{St} \mapsto \wp(\overline{Sync})$
- Multithreaded state:

$$\overline{\Psi} : \overline{TId} \to \overline{St}^{\vec{+}}$$
$$\overline{\Omega} : \overline{TId} \to ((\overline{TId} \times Integer) \cup \overline{\bot}_{\Omega})$$

Intra-thread semantics

- *visible*$^{\#}$ function: straight abstraction of the concrete one!
- *step*$^{\#}$ function: defines a single step in the computation. Makes the upper bound between all the visible values $\rightarrow$ deterministic choice!

Definition (Single-thread semantics $\overline{\mathbb{S}}^{\circ}$)

$$\begin{aligned}
\overline{\mathbb{S}}^{\circ} &: (\overline{\Psi} \times \overline{\Omega} \times \overline{TId}) \mapsto \overline{St}^{\vec{+}} \\
\overline{\mathbb{S}}^{\circ} &[\![\bar{\mathsf{f}}, \bar{\mathsf{r}}, \bar{\mathsf{t}}]\!] = lfp_{\emptyset}^{\sqsubseteq} \lambda\overline{\tau}.\{\overline{\sigma}_0\} \sqcup_{\tau} \{\overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} \rightarrow \overline{\sigma}_i : \\
&\qquad \overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_{i-1} = \overline{\tau} \wedge \overline{\sigma}_i = step^{\#}(\bar{\mathsf{t}}, \bar{\mathsf{f}}, \bar{\mathsf{r}})\}
\end{aligned}$$

$step^{\#}$

Definition ($step^{\#}$ function)

$$
\begin{aligned}
step^{\#} \quad & : \overline{TId} \times \overline{\Psi} \times \overline{\Omega} \mapsto \overline{St} \\
step^{\#} \quad & (\bar{t}, \bar{f}, \bar{s}) = \sigma \text{ where } \bar{f}(\bar{t}) = \overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_i \text{ and} \\
& \overline{\sigma}_i \overset{\circ}{\rightarrow}{}^{\#} \overline{\sigma} \qquad \text{if } \pi_1(action^{\#}(\overline{\sigma}_i)) \neq \mathtt{r} \\
& \\
& \overline{\sigma}'_i \overset{\circ}{\rightarrow}{}^{\#} \overline{\sigma} : \qquad \text{if } \pi_1(action^{\#}(\overline{\sigma}_i)) = \mathtt{r} \\
& \overline{V} = visible^{\#}(\bar{t}, \pi_2(action^{\#}(\overline{\sigma}_i)), synchronised^{\#}(\overline{\sigma}_i), \bar{f}, \bar{s}(\bar{t})) \\
& \bar{v} = \bigsqcup_{\bar{v}' \in \overline{V}}{}_{Val}\, \bar{v}', \bar{l} = action^{\#}(\overline{\sigma}_i) \\
& \overline{sh} = shared^{\#}(\overline{\sigma}_i), \overline{sh}' = \overline{sh}[\bar{l} \mapsto \bar{v} \sqcup_{Val} \overline{sh}(\bar{l})] \\
& \overline{\sigma}'_i = set_shared^{\#}(\overline{\sigma}, \overline{sh}')
\end{aligned}
$$

Multithread semantics

- Each time we compute the single thread semantics of a thread it may expose new values to other threads
- Iterates the single-thread semantics until a fixpoint is reached
- ... as in the concrete semantics!

Definition (Multithreaded semantics $\overline{\mathbb{S}}^{\|}$)

$$\overline{\mathbb{S}}^{\|} : \overline{\Psi} \times \overline{\Omega} \mapsto \overline{\Psi} \times \overline{\Omega}$$

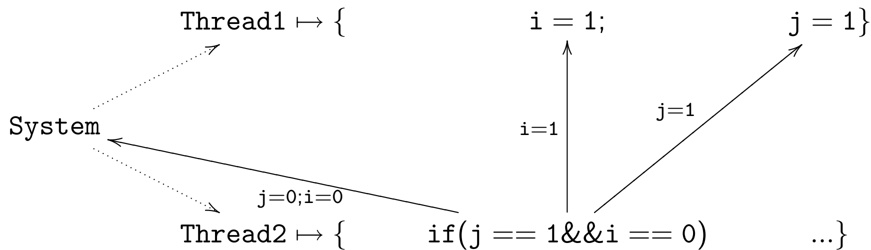
$$\overline{\mathbb{S}}^{\|} [\![\bar{f}_0, \bar{r}_0]\!] = lfp_{\emptyset}^{\sqsubseteq} \lambda(\bar{f}, \bar{r}).\{(\bar{f}_0, \bar{r}_0)\} \sqcup_f \{(\bar{f}_i, \bar{r}) : \forall \bar{t} \in dom(\bar{f}) : \bar{f}_i(\bar{t}) = \overline{\mathbb{S}}^{\circ} [\![\bar{f}, \bar{t}]\!]\}$$

Correctness

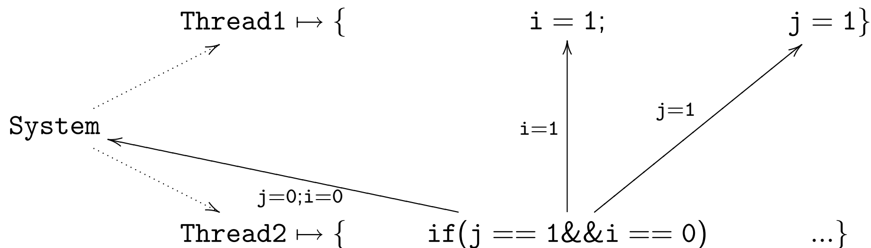
We formally proved the correctness of our analysis, i.e. :

Theorem (Soundness of $\overline{\mathbb{S}}^{\|}$)
$\forall \bar{\mathrm{f}} \in \overline{\Psi}_{pre} : \alpha_f(\mathbb{S}^{\|})[\![\bar{\mathrm{f}}]\!] \sqsubseteq_f \overline{\mathbb{S}}^{\|}[\![\bar{\mathrm{f}}]\!]$.

An example - 3rd iteration

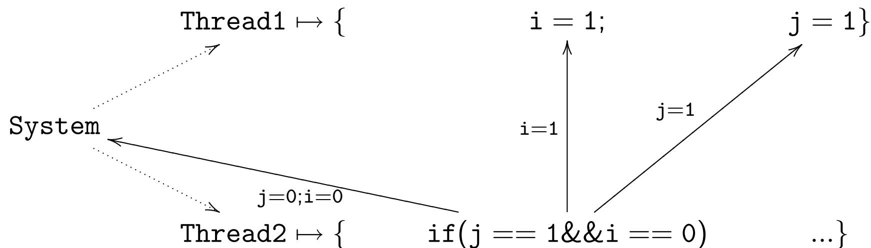


An example - 3rd iteration



$$visible^{\#}(\mathtt{j}) = [0..0] \sqcup [1..1] = [0..1]$$

An example - 3rd iteration



$visible^{\#}(\mathtt{j}) = [0..0] \sqcup [1..1] = [0..1]$
$visible^{\#}(\mathtt{i}) = [0..0] \sqcup [1..1] = [0..1]$

Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

From theory to practice

- Goal: Apply the theoretical framework to a real programming language - Java

From theory to practice

- Goal: Apply the theoretical framework to a real programming language - Java
- We talked about shared memory, threads, monitors...
  - Shared memory: the heap
  - Threads: objects
  - Monitors: defined on objects

From theory to practice

- Goal: Apply the theoretical framework to a real programming language - Java
- We talked about shared memory, threads, monitors...
  - Shared memory: the heap
  - Threads: objects
  - Monitors: defined on objects
- In addition:
  - Objects stored in the heap
  - Heap relates references to objects

From theory to practice

- Goal: Apply the theoretical framework to a real programming language - Java
- We talked about shared memory, threads, monitors...
  - Shared memory: the heap
  - Threads: objects
  - Monitors: defined on objects
- In addition:
  - Objects stored in the heap
  - Heap relates references to objects
- Conclusion:
  - Shared memory accessed by reference
  - Threads identified by reference
  - Monitors identified by reference

Alias analysis

- "Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way. Two pointers are said to be aliased if they point to the same location." (Wikipedia)

Alias analysis

- "Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way. Two pointers are said to be aliased if they point to the same location." (Wikipedia)
- About the shared memory:
  - Check if two accesses MAY be on the same location
  - i.e.: Check if two pointers MAY point to the same location

Alias analysis

- "Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way. Two pointers are said to be aliased if they point to the same location." (Wikipedia)
- About the shared memory:
  - Check if two accesses MAY be on the same location
  - i.e.: Check if two pointers MAY point to the same location
- About the monitors:
  - Check if two locks (performed by different threads) are ALWAYS on the same monitors
  - i.e.: Check if two pointers ALWAYS point to the same location

Alias analysis

- "Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way. Two pointers are said to be aliased if they point to the same location." (Wikipedia)
- About the shared memory:
  - Check if two accesses MAY be on the same location
  - i.e.: Check if two pointers MAY point to the same location
- About the monitors:
  - Check if two locks (performed by different threads) are ALWAYS on the same monitors
  - i.e.: Check if two pointers ALWAYS point to the same location
- Conclusion: a combination of the may- and must- alias analysis

Solution - An intuition

- A may-aliasing domain that:
  - Relates abstract references to the point of the program that creates it
  - Finite (as the statements of the program are finite)
  - Used to analyze the accesses to shared memory

Solution - An intuition

- A may-aliasing domain that:
  - Relates abstract references to the point of the program that creates it
  - Finite (as the statements of the program are finite)
  - Used to analyze the accesses to shared memory
- A must-aliasing domain that:
  - Relates abstract references to an equivalence class (two abstract references are always equal if they are related to the same equivalence class)
  - Used to identify monitors, i.e. to lock and unlock them

Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

Bytecode

```
public void prova(int i) {
 if(i >= 0)
  System.out.println("Ok");
 else System.out.println("No");
}
```

Bytecode

```
public void prova(int i) {
 if(i >= 0)
  System.out.println("Ok");
 else System.out.println("No");
}
```

```
 0 iload_1
 1 iflt 15 (+14)
 4 getstatic #2 <java/lang/System.out>
 7 ldc #3 <Ok>
 9 invokevirtual #4 <java/io/PrintStream.println>
12 goto 23 (+11)
15 getstatic #2 <java/lang/System.out>
18 ldc #5 <No>
20 invokevirtual #4 <java/io/PrintStream.println>
23 return
```

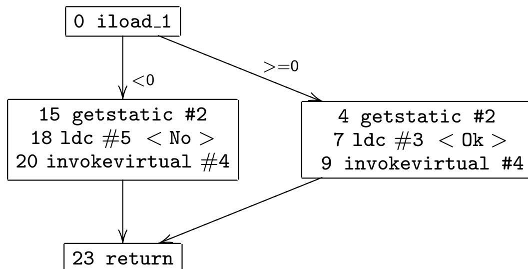
Bytecode

```
0 iload_1
1 iflt 15 (+14)
4 getstatic #2 < java/lang/System.out >
7 ldc #3 < Ok >
9 invokevirtual #4 < java/io/PrintStream.println >
12 goto 23 (+11)
15 getstatic #2 < java/lang/System.out >
18 ldc #5 < No >
20 invokevirtual #4 < java/io/PrintStream.println >
23 return
```

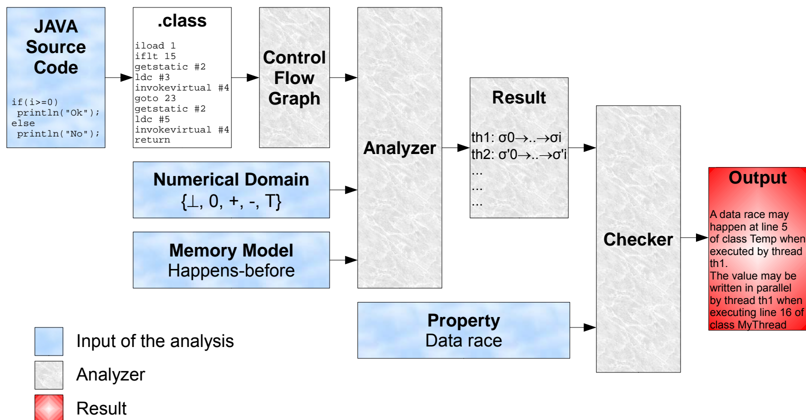
Bytecode

```
 0 iload_1
 1 iflt 15 (+14)
 4 getstatic #2 < java/lang/System.out >
 7 ldc #3 < Ok >
 9 invokevirtual #4 < java/io/PrintStream.println >
12 goto 23 (+11)
15 getstatic #2 < java/lang/System.out >
18 ldc #5 < No >
20 invokevirtual #4 < java/io/PrintStream.println >
23 return
```

⇓



Structure



Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

Implementation

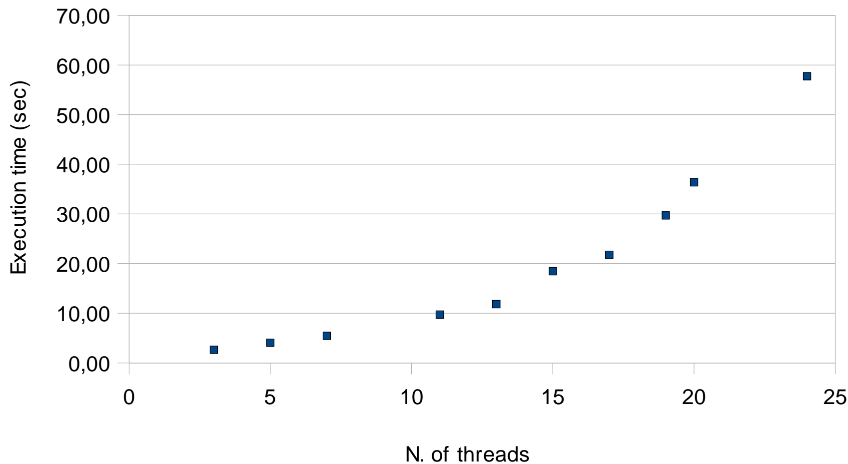
- We have implemented the analysis
  - analyze (a subset of) Java bytecode
  - developed in Java
- Tested on a set of 10 applications built ad hoc
- Executed on an AMD Athlon 64 3000+ processor with 2 GB of RAM running a GNU/Linux operating system with the Java virtual machine version 1.5.0
- The analysis has been executed 10 times on each application and we take the average on the times of execution

Results

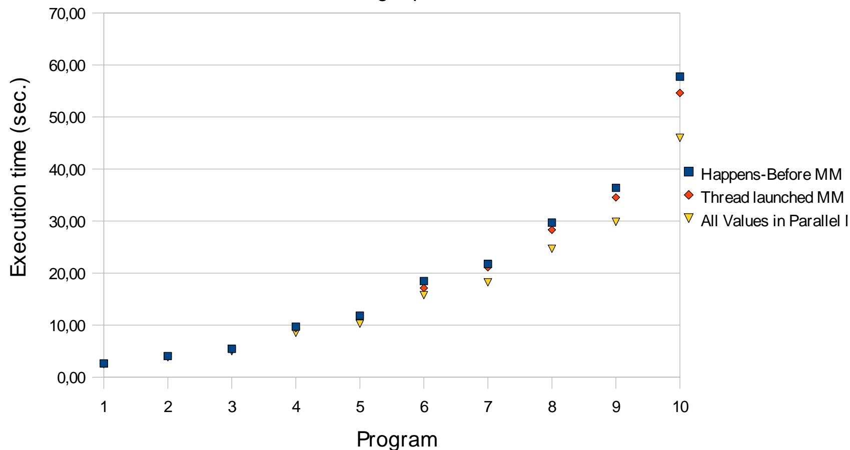
# Pr.	# Th.	# St.	HB (sec)	TS (sec)	AP (sec)
1	3	424	2,63	2,58	2,51
2	5	637	4,06	4,03	3,81
3	7	751	5,44	5,32	4,99
4	11	977	9,70	9,41	8,53
5	13	1.097	11,83	11,83	10,26
6	15	1.310	18,47	17,14	15,77
7	17	1.422	21,75	21,11	18,22
8	19	1.635	29,70	28,30	24,67
9	20	1.742	36,37	34,54	29,86
10	24	2.126	57,76	54,62	45,98

Table: Experimental results

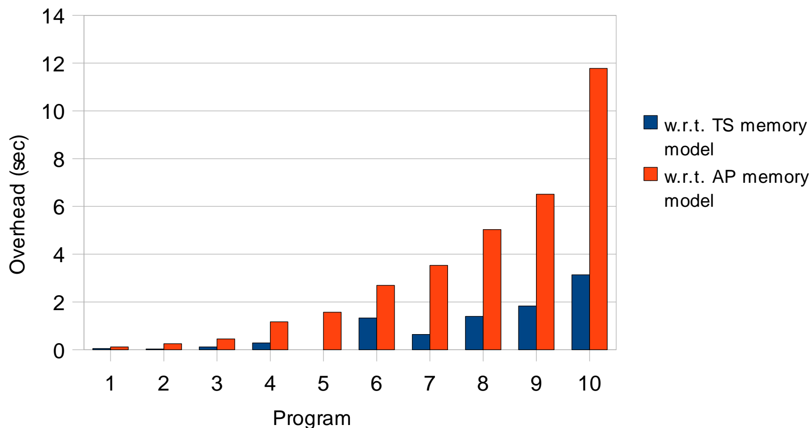
Execution time vs. N. of threads



Memory models



Overhead



Outline

Memory Model

Concrete domain and semantics

Abstraction

Java

The analyzer

Demo

Experimental results

Conclusion

Related works

- Many papers have been focused on the analysis of specific properties like data race condition [RIN01]. Usually these approaches suppose that the executions are sequentially consistent
- Other papers define memory models [MAN05,CEN07,SAR07] but they do not propose any static analysis
- Two model checkers [ROY02,HUY06] sound w.r.t. a memory model have been proposed in the last years. They are afflicted by the state explosion problem, and they seem not in position to scale up

Conclusion

- We formalize in fixpoint form the happens-before memory model
  - generic w.r.t. the programming language
  - supports mutual exclusion
- We abstract it
  - The main characteristic components of the memory model are straightly abstracted!
    - For the next memory model, define the concrete semantics and obtain the abstract one!
- We prove formally the soundness of our abstraction
- We implement it and the preliminary experimental results are quite encouraging
- As far as we know, it is the first static analysis sound w.r.t. the happens-before memory model

Publications

- Concrete and abstract semantics of the happens-before memory model: **P. Ferrara** *"Static analysis via abstract interpretation of the happens-before memory model"*, in Springer, editor, Proceedings of the Second International Conference on Tests and Proofs (TAP 2008), volume 4966 of *Lecture Notes in Computer Science*, Prato, Italy, April 9-11, 2008
- Alias analysis and a previous implementation applied to the data race detection:
- **Pietro Ferrara** *"A fast and precise analysis for data race detection"*, in Elsevier, editor, Proceedings of Bytecode 08, volume *ENTCS*, Budapest, Hungary, April 5, 2008
- In addition: new implementation and experimental results

Bibliography - Memory Models and Concurrency

[LAM79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In IEEE Trans. Computers, volume 28, pages 690691, 1979.

[LAM78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In Commun. ACM, volume 21, pages 558565, New York, NY, USA, 1978. ACM Press.

[MAN05] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In ACM Press, editor, Proceedings of POPL 05, 2005.

[LEE06] Edward A. Lee. The problem with threads. In IEEE Computer, volume 39, pages 3342, Los Alamitos, CA, USA, May 2006. IEEE Computer Society Press.

[SUT05] Herb Sutter and James Larus. Software and the concurrency revolution. In ACM Queue, Vol. 3, No. 7, pages 5462, September 2005.

Bibliography - Related Works

[RIN01] Martin C. Rinard. Analysis of multithreaded programs. In SAS 01, pages 119, London, UK, 2001. Springer-Verlag.

[CEN07] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The java memory model: Operationally, denotationally, axiomatically. In Springer Verlag, editor, Proceeding of ESOP 07, number 4421 in Lecture Notes in Computer Science, pages 331346, 2007.

[SAR07] Vijay Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. To appear (http://www.saraswat.org/raofull.pdf), 2006.

[ROY02] Abhik Roychoudhury and Tulika Mitra. Specifying multithreaded java semantics for program verification. In ACM Press, editor, Proceedings of ICSE, May 2002.

[HUY06] Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for c#. In Springer-Verlag, editor, FM, volume 4085 of Lecture Notes in Computer Science, 2006.

Definition (*visible* function)

$$
\begin{aligned}
&\mathit{visible} : \mathit{TId} \times \mathit{Loc} \times \wp(\mathit{Sync}) \times \Psi \times ((\mathit{TId} \times \mathit{Integer}) \cup \bot_\Omega) \mapsto \wp(\mathit{Val}) \\
&\mathit{visible}(t, l, S, f, (t', i')) = \\
&(1) = \mathit{project}(l, \mathit{suffix}(f(t'), i'), S) \cup \\
&(2) \quad \{v : v \in \mathit{project}(l, f(t''), S) : t'' \in \mathit{dom}(f) \setminus \{t, t'\}\}
\end{aligned}
$$

Definition (*suffix* function)

$$\begin{aligned} \mathit{suffix} &: \vec{St^{+}} \times \mathit{Integer} \mapsto \vec{St^{+}} \\ \mathit{suffix} &\quad (\sigma_0 \rightarrow \cdots \rightarrow \sigma_j, i) = \begin{cases} \sigma_i \rightarrow \cdots \rightarrow \sigma_j & \text{if } i \geq 0 \wedge i < j \\ \epsilon & \text{if } i = j \end{cases} \end{aligned}$$

Definition (*project* function)

$$
\begin{aligned}
\textit{project} \quad & : \textit{Loc} \times \textit{St}^{\vec{+}} \times \wp(\textit{Sync}) \mapsto \wp(\textit{Val}) \\
\textit{project} \quad & (l, \sigma_0 \to \cdots \to \sigma_i, S) = \{v : \exists j \in [0..i] : \\
& \textit{action}(\sigma_j) = (\mathtt{w}, l, v) \wedge \textit{not_synchronised}(\sigma_j \to \cdots \to \sigma_i, S)\}
\end{aligned}
$$

Definition (*not_synchronised* function)

$$
\begin{array}{l}
\mathit{not_synchronised} : St^{\vec{+}} \times \wp(Sync) \mapsto \{\text{true}, \text{false}\} \\
\mathit{not_synchronised}(\sigma_0 \rightarrow \cdots \rightarrow \sigma_i, S) = \text{true if and only if} \\
(1) S \cap \mathit{synchronised}(\sigma_0) = \emptyset \vee \\
(2) \nexists \sigma_j \in \mathit{cut}(\sigma_0 \rightarrow \cdots \rightarrow \sigma_i, S) : \mathit{action}(\sigma_j) = (\mathtt{w}, l, v), \\
\qquad \mathit{action}(\sigma_0) = (\mathtt{w}, l_0, v_0), l = l_0
\end{array}
$$

Definition (*cut* function)

$$\begin{aligned} \mathit{cut} &: \mathit{St}^{\vec{+}} \times \wp(\mathit{Sync}) \mapsto \mathit{St}^{\vec{+}} \\ \mathit{cut} &\ (\sigma_0 \to \cdots \to \sigma_i, S) = \begin{cases} \epsilon \text{ if } \mathit{synchronised}(\sigma_0) \cap S = \emptyset \\ \sigma_0 \to \mathit{cut}(\sigma_1 \to \cdots \to \sigma_i, S) \text{ otherwise} \end{cases} \end{aligned}$$

Definition ($step^{\#}$ function)

$$step^{\#} \; : \overline{TId} \times \overline{\Psi} \times \overline{\Omega} \mapsto \overline{St}$$

$$step^{\#} \; (\bar{t}, \bar{f}, \bar{s}) = \sigma \text{ where } \bar{f}(\bar{t}) = \overline{\sigma}_0 \to \cdots \to \overline{\sigma}_i \text{ and}$$

$$\overline{\sigma}_i \overset{\circ}{\to}^{\#} \overline{\sigma} \qquad \text{if } \pi_1(action^{\#}(\overline{\sigma}_i)) \neq \mathtt{r}$$

$$\overline{\sigma}'_i \overset{\circ}{\to}^{\#} \overline{\sigma} : \qquad \text{if } \pi_1(action^{\#}(\overline{\sigma}_i)) = \mathtt{r}$$

$$\overline{V} = visible^{\#}(\bar{t}, \pi_2(action^{\#}(\overline{\sigma}_i)), synchronised^{\#}(\overline{\sigma}_i), \bar{f}, \bar{s}(\bar{t}))$$

$$\bar{v} = \bigsqcup_{\bar{v}' \in \overline{V}} {}_{Val} \; \bar{v}'$$

$$\overline{sh} = shared^{\#}(\overline{\sigma}_i), \overline{sh}' = \overline{sh}[\bar{l} \mapsto \bar{v} \sqcup_{Val} \overline{sh}(\bar{l})]$$

$$\overline{\sigma}'_i = set_shared^{\#}(\overline{\sigma}, \overline{sh}')$$

Definition (*visible*$^{\#}$ function)

$$
\begin{aligned}
\mathit{visible}^{\#} &: \overline{\mathit{TId}} \times \overline{\mathit{Loc}} \times \wp(\overline{\mathit{Sync}}) \times \overline{\Psi} \times (\overline{\mathit{TId}} \times \mathit{Integer}) \mapsto \wp(\overline{\mathit{Val}}) \\
\mathit{visible}^{\#} & (\bar{\mathrm{t}}, \bar{\mathrm{l}}, \overline{\mathrm{S}}, \bar{\mathrm{f}}, (\bar{\mathrm{t}}', i')) = \\
& = \mathit{project}^{\#}(\bar{\mathrm{l}}, \mathit{suffix}^{\#}(\bar{\mathrm{f}}(\bar{\mathrm{t}}'), i'), \overline{\mathrm{S}}) \cup \\
& \{\bar{\mathrm{v}} : \bar{\mathrm{v}} \in \mathit{project}^{\#}(\bar{\mathrm{l}}, \bar{\mathrm{f}}(\bar{\mathrm{t}}''), \overline{\mathrm{S}}) : \bar{\mathrm{t}}'' \in \mathit{dom}(\bar{\mathrm{f}}) \setminus \{\bar{\mathrm{t}}, \bar{\mathrm{t}}'\}\}
\end{aligned}
$$

Definition (*suffix*$^{\#}$ function)

$$
\begin{aligned}
\mathit{suffix}^{\#} &: \overline{St}^{\vec{+}} \times \mathit{Integer} \mapsto \overline{St}^{\vec{+}} \\
\mathit{suffix}^{\#} &\ (\overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_j, i) = \begin{cases} \overline{\sigma}_i \rightarrow \cdots \rightarrow \overline{\sigma}_j & \text{if } i \geq 0 \wedge i < j \\ \overline{\epsilon} & \text{if } i = j \end{cases}
\end{aligned}
$$

Definition (*project*$^{\#}$ function)

$$
\begin{array}{l}
project^{\#} : \overline{Loc} \times \overline{St}^{\vec{+}} \times \wp(\overline{Sync}) \mapsto \wp(\overline{Val}) \\
project^{\#}(\bar{\mathsf{l}}, \overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_i, \bar{\mathsf{S}}) = \{\bar{\mathsf{v}} : \exists j \in [0..i] : \\
\quad action^{\#}(\overline{\sigma}_j) = (\mathtt{w}, \bar{\mathsf{l}}, \bar{\mathsf{v}}) \wedge not_synchronised^{\#}(\overline{\sigma}_j \rightarrow \cdots \rightarrow \overline{\sigma}_i, \bar{\mathsf{S}})\}
\end{array}
$$

Definition (*not_synchronised*$^{\#}$ function)

$$\begin{array}{l} \mathit{not_synchronised}^{\#} : \overline{St}^{\vec{+}} \times \wp(\overline{Sync}) \mapsto \{\mathsf{true}, \mathsf{false}\} \\ \mathit{not_synchronised}^{\#}(\overline{\sigma}_0 \to \cdots \to \overline{\sigma}_i, \overline{\mathsf{S}}) = \mathsf{true} \text{ if and only if} \\ \quad \overline{\mathsf{S}} \cap \mathit{synchronised}^{\#}(\overline{\sigma}_0) = \emptyset \vee \\ \quad \nexists \overline{\sigma}_j \in \mathit{cut}^{\#}(\overline{\sigma}_0 \to \cdots \overline{\sigma}_i, \overline{\mathsf{S}}) : \\ \quad \mathit{action}^{\#}(\overline{\sigma}_{j-1}) = (\mathtt{w}, \bar{\mathsf{l}}, \bar{\mathtt{v}}), \mathit{action}^{\#}(\overline{\sigma}_0) = (\mathtt{w}, \bar{\mathsf{l}}_0, \bar{\mathtt{v}}_0), \bar{\mathsf{l}} = \bar{\mathsf{l}}_0 \end{array}$$

Definition ($cut^{\#}$ function)

$$
\begin{array}{l}
cut^{\#} : \overline{St}^{\vec{+}} \times \wp(\overline{Sync}) \mapsto \overline{St}^{\vec{+}} \\
cut^{\#}(\overline{\sigma}_0 \rightarrow \cdots \rightarrow \overline{\sigma}_i, \overline{\mathsf{S}}) = \\
\quad = \begin{cases} \epsilon \text{ if } synchronised^{\#}(\overline{\sigma}_0) \cap \overline{\mathsf{S}} = \emptyset \\ \overline{\sigma}_0 \rightarrow cut^{\#}(\overline{\sigma}_1 \rightarrow \cdots \rightarrow \overline{\sigma}_i, \overline{\mathsf{S}}) \text{ otherwise} \end{cases}
\end{array}
$$