A fast and precise analysis for data race detection

Pietro Ferrara

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

April 5, 2008

Abstract interpretation

- Static analysis
- Mathematical theory to define and soundly approximate the semantics of a program [CC77,CC79]
- Concrete domain: $\langle \mathsf{A}, \sqsubseteq_\mathsf{C}, \bot_\mathsf{C}, \top_\mathsf{C}, \sqcup_\mathsf{C}, \sqcap_\mathsf{C} \rangle$
- The abstract domain approximates it $\langle \widehat{\mathsf{A}}, \sqsubseteq_\mathsf{A}, \bot_\mathsf{A}, \top_\mathsf{A}, \sqcup_\mathsf{A}, \sqcap_\mathsf{A} \rangle$
- Abstraction α and concretization γ functions:
  - $\forall \mathsf{s} \in \mathsf{A} : \mathsf{s} \sqsubseteq_\mathsf{C} \gamma(\alpha(\mathsf{s}))$
  - $\forall \widehat{\mathsf{s}} \in \widehat{\mathsf{A}} : \alpha(\gamma(\widehat{\mathsf{s}})) \sqsubseteq_\mathsf{A} \widehat{\mathsf{s}}$
- Abstract and concrete semantics:

$$\forall \mathsf{s} \in \mathsf{A} : \mathbb{F}[\![\mathsf{s}]\!] \sqsubseteq_\mathsf{C} \gamma(\widehat{\mathbb{F}}[\![\alpha(\mathsf{s})]\!])$$

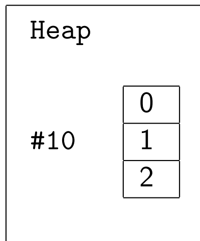
Java and Multithreading

- Java: native support for threads
- Threads are objects $\rightarrow$ identified by reference
- `Thread.start()` launches a new thread
- `monitorenter` and `monitorexit`
- Many other ways of synchronizing
  - `wait` and `notify` on objects
  - `isAlive()` on threads
  - ...
- We focus only on monitors

Data Race Condition

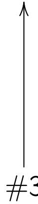
- "two threads concurrently access the same area of shared memory, they are not synchronized, and at least one is performing a write operation"
- data race $\approxeq$ a bug
- In Java:
  - Shared memory $\rightarrow$ heap
  - Monitors $\rightarrow$ objects
  - Threads $\rightarrow$ objects

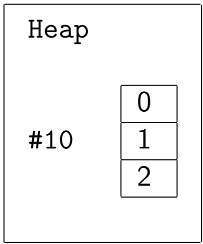
```
Thread1:      ··· putfield #1
```




```
Thread2:      ··· putfield #1
```

Thread1: ··· putfield #1

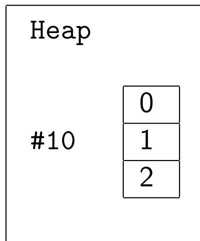




Thread2: ··· putfield #1

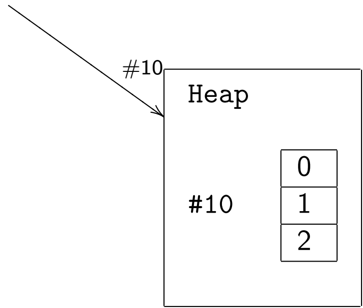
Thread1: $\cdots$ putfield #1

#8



Thread2: $\cdots$ putfield #1

Thread1: $\cdots$ `putfield #1`



Thread2: $\cdots$ `putfield #1`

Thread1: ⋯ putfield #1

#10

Heap

#10

0
1
2

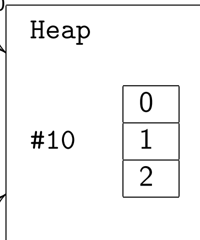
#10

Thread2: ⋯ putfield #1

{#15}

Thread1: ··· putfield #1

#10



#10

Thread2: ··· putfield #1

{#12}

Concrete domain

- We need to precisely trace the accesses to the shared memory
- At low level
- In practice, formalization of the Java Virtual Machine specification:
  - Values: references or integers - $\mathsf{Val} = \mathsf{Ref} \cup \mathbb{N}$
  - Operand stack: stack of values - $\mathsf{Op} = \mathbb{ST}(\mathsf{Val})$
  - Local Variables: array of values - $\mathsf{LV} = \mathbb{AR}(\mathsf{Val})$
  - Frame: an operand stack, a local variables element, a program counter - $\mathsf{F} = \mathsf{Op} \times \mathsf{LV} \times \mathsf{PC}$

Monitors

- Single thread state
- Monitors can be locked more than once
- Function that traces for each monitor how many times has been locked

Definition (Monitor)

$\mathsf{L} = [\mathsf{Ref} \rightarrow \mathbb{N}]$

Heap

- Heap: reference $\mapsto$ local variables' element
- We augment the information contained by the local variables
- For each value, we trace
  - The thread that wrote it, identified by its reference
  - The program counter of the instruction
  - The stack of called method
  - The owned monitors
- Necessary in order to check the data race condition

Definition (Augmented Local Variables)

$\mathsf{ALV} = \mathbb{AR}(\mathsf{Val} \times \mathsf{Ref} \times \mathsf{PC} \times \mathbb{ST}(\mathsf{PC}) \times \mathsf{L})$

Definition (Heap)

$\mathsf{H} = [\mathsf{Ref} \rightarrow \mathsf{ALV}]$

Monothread state

- Monothread state:
  - A stack of frame (one for each method call)
  - An heap
  - A monitors' function

Definition (State)

$\Sigma = \mathbb{ST}(\mathsf{F}) \times \mathsf{H} \times \mathsf{L}$

Multithreaded state

- Collects for each thread the trace of its execution
- The inter-thread order of execution is abstracted away
- The same approach has been adopted by [FER08]

Definition (Multithreaded State)

$\Omega = \wp([\mathsf{Ref} \rightarrow \vec{\Sigma^{+}}])$

Concrete semantics

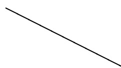
- Partial trace semantics [CC79]
- At each step
  - it chooses randomly a thread
  - it checks if it is going to acquire a monitor already owned by another thread
  - it takes its last state
  - it builds up a possible state of the heap following a memory model [FER08]
  - it executes one step
  - it stores the result
- Until a fixpoint is reached

Data Race Condition

- Check **locally** if a data race happens
- When a value in the shared memory is accessed:
  - check if it has been written by another thread
  - check if it owned a monitor currently acquired by the thread that is performing the access
  - if it is not the case, a data race is detected

$\{[\#15 \mapsto 1]\}$

`Thread1:` $\cdots$ `putfield #1`



$\#3$

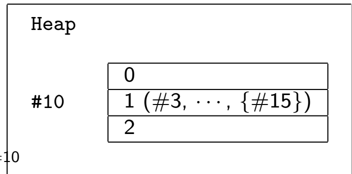
$\#10$

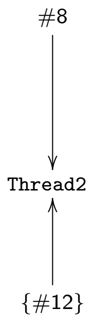
`Heap`

`#10`	
	0
	1 $(\#3, \cdots, \{\#15\})$
	2

`Thread2:` $\cdots$ `putfield #1`

`Thread1:` $\cdots$ `putfield #1`





`Thread2:` $\cdots$ `putfield #1`

$\#3 \neq \#8$

$\{\#15\} \cap \{\#12\} = \emptyset$

`Data Race!`

Abstract domain

$$\widehat{\mathsf{Val}} = \widehat{\mathsf{Ref}} \cup \hat{\mathsf{i}}$$
$$\widehat{\mathsf{Op}} = \mathbb{ST}(\widehat{\mathsf{Val}})$$
$$\widehat{\mathsf{LV}} = \mathbb{AR}(\widehat{\mathsf{Val}})$$
$$\hat{\mathsf{F}} = \widehat{\mathsf{Op}} \times \widehat{\mathsf{LV}} \times \mathsf{PC}$$

Alias analysis

- Alias analysis that traces
  - When two values **may** point to the same address
    - check when two threads may access the same area of shared memory
  - When two values contains **always** the same address
    - check when two threads are surely synchronized on the same monitor

May-aliasing

- Abstract reference:
  - the program counter
  - the stack of the called methods
  - the abstract reference of the executing thread

  when the `new` statement allocates it
- #mainthread
- An abstract reference represents multiple concrete references
- Used to
  - Identify threads
  - Check when two accesses to the shared memory may be on the same location

Definition (May-alias domain)

$\widehat{\mathsf{P}} = (\mathsf{PC} \times \mathbb{ST}(\mathsf{PC}) \times \widehat{\mathsf{P}}) \cup \{\#\text{mainthread}\}$

Must-aliasing

- Equivalence classes
- Identify monitors
- If two threads lock on the same equivalence class they are synchronized

Definition (Monitors)

$\widehat{\mathsf{L}} = [\widehat{\mathsf{E}} \to \mathbb{N}]$

Definition

$\widehat{\mathsf{Ref}} = \widehat{\mathsf{E}} \times \wp(\widehat{\mathsf{P}})$

Augmented local variables

- In the concrete, different executions may contain at the same point of computation
  - different values
  - written by different threads and different points of the program
- In abstract
  - collect together all these values
  - lub of all possible values
  - set union of all the threads and program points that may write the value

Definition

$\widehat{\mathsf{ALV}} = \mathbb{AR}(\widehat{\mathsf{Val}} \times \wp(\widehat{\mathsf{P}} \times \mathsf{PC} \times \mathbb{ST}(\mathsf{PC}) \times \widehat{\mathsf{L}}))$

Definition (Heap)

$\widehat{\mathsf{H}} = [\widehat{\mathsf{P}} \to \widehat{\mathsf{ALV}}]$

Multithreaded state

Definition (Monothread State)

$$\widehat{\Sigma} = \mathbb{ST}(\widehat{\mathsf{F}}) \times \widehat{\mathsf{H}} \times \widehat{\mathsf{L}}$$

Definition (Multithreaded State)

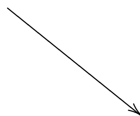
$$\widehat{\Omega} = [\widehat{\mathsf{P}} \to \widehat{\Sigma}^{\vec{+}}]$$

Abstract semantics

- Computed through two nested fixpoints
  - Single thread semantics: approximate the semantics of a single thread given a multithreaded state
    - values written in parallel by other threads
  - Multithread semantics: iterate for all the active threads their single thread semantics passing at each iteration the last multithread state
- This approach has been formalized and proved to be sound on the happens-before memory model by a previous work [FER08]

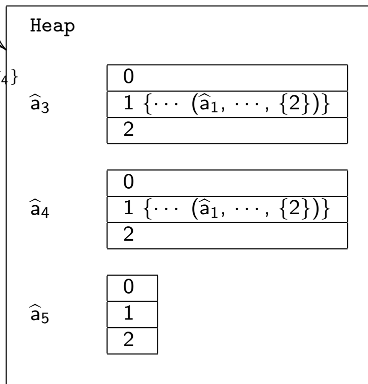
$\{[2 \mapsto 1]\}$

`Thread1:` $\cdots$ `putfield #1`



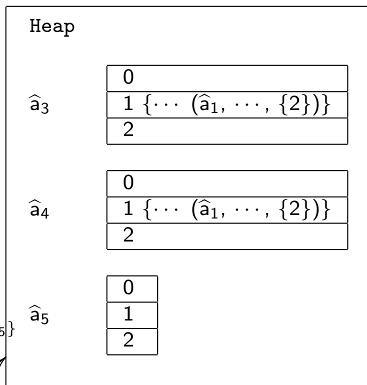
$\widehat{a}_1$

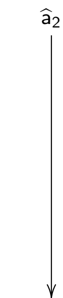
$\{\widehat{a}_3, \widehat{a}_4\}$



`Thread2:` $\cdots$ `putfield #1`

Thread1: $\cdots$ putfield #1





$\{\widehat{a}_4, \widehat{a}_5\}$

Thread2: $\cdots$ putfield #1

$\{18 \mapsto 1\}$

$\widehat{a}_2 \neq \widehat{a}_1$

$\{2\} \cap \{18\} = \emptyset$

Data Race!

If a data race occurs

- Data race $\Rightarrow$ An access to the memory reads or overwrites a value written by another thread
- We know:
  - Program points that cause the data race
  - The call stack followed to arrive at that point
  - The threads that perform the accesses
  - Equivalence classes of the monitors owned when the value has been previously written by the other thread
- Equivalence classes reachable from the local environment
- Suggest which objects should lock (we can have 0, 1, or more suggestions!)
- Only an advice

# Application	# Threads	# Statements	Time (sec)
1	3	424	7"
2	5	637	7"
3	7	751	10"
4	11	977	13"
5	13	1.097	17"
6	15	1.310	24"
7	17	1.422	25"
8	19	1.635	36"
9	20	1.742	49"
10	24	2.126	1'07"

Table: Experimental results

Discussion

- Encouraging
- Test our analysis with bigger (i.e. industrial) examples
- No false alarms

Conclusion

- New static analysis of data races
- Combination of a may- and a must- alias analysis
- Provide useful information when a data race may happen in order to discover missing lock actions
- Implemented, the experimental results are promising

Future works

- Extend the analyzer in order to support all the Java bytecode language
- Apply it to bigger and perhaps industrial programs
- Investigate other properties on multithreaded programs, like determinism

Bibliography

[FER08] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In Springer, editor, Proceedings of TAP 08, LNCS, 2008.

Bibliography - Abstract Interpretation

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL 77, pages 238-252, Los Angeles, California, 1977. ACM Press.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In POPL 79, pages 269-282, San Antonio, Texas, 1979. ACM Press.

Bibliography - Alias Analysis

[DEU94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In ACM Press, editor, Proceedings of PLDI 04, 1994.

[VEN02] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In Springer, editor, SAS, LNCS, 2002.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3valued logic. In ACM Press, editor, Proceedings of POPL 99, 1999.

Bibliography - Data Race

[BAN03] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling data race detection in the intel thread checker. In Proceedings of STMCS 06, 2006.

[CAL03] R. OCallahan and Jong-Deok Choi. Hybrid dynamic data race detection. In ACM Press, editor, Proceedings of PPoPP 03, 2003.

[POZ03] E. Poznianski and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In IEEE Computer Society, editor, Proceedings of IPDPS 03, 2003.

[SAV97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In ACM Press, editor, Proceedings of SOSP 97, 1997.

[YU05] Y. Yu, T. Rodeheer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In ACM Press, editor, Proceedings of SOSP 05, 2005.

[ABA06] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. In ACM Press, editor, Proceedings of TOPLAS 06, 2006.