#### "Context-sensitive Points-to Analysis: Is it worth it?"

by Ondřej Lhoták and Laurie Hendren

Presented by Nicholas Rudzicz

#### Question

Is context-sensitive points-to analysis worth it?

#### Question

- Is context-sensitive points-to analysis worth it?
  - Yes.

#### The End



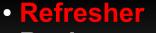
#### Outline

- Course refresher
  - Motivation
- Background
  - Pedantic stuff
- for(int i=0; i < num\_benchmarks; i++)</p>
  - 4 describe\_benchmark(i);
  - discuss\_results\_of\_benchmark(i); }
- Conclusions



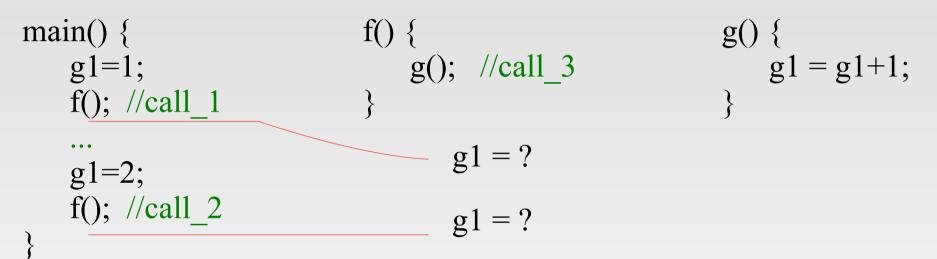
- Background
- Benchmarks (0/7)
- Conclusion

Recall this example (without pointers) from class:

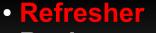


- Background
- Benchmarks (0/7)
- Conclusion

Recall this example (without pointers) from class:

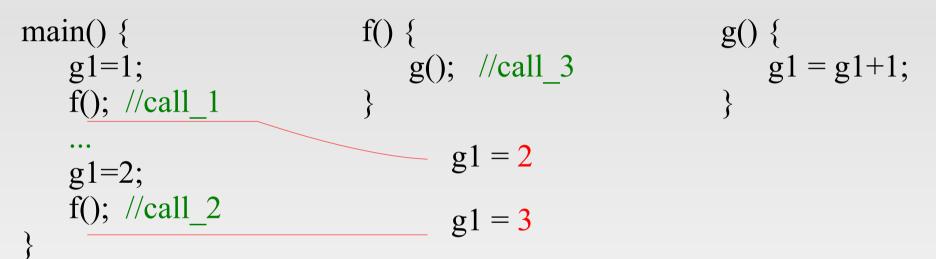


- Interprocedural analysis
  - W/out context, know nothing about g1 after calls



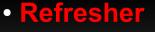
- Background
- Benchmarks (0/7)
- Conclusion

Recall this example (without pointers) from class:



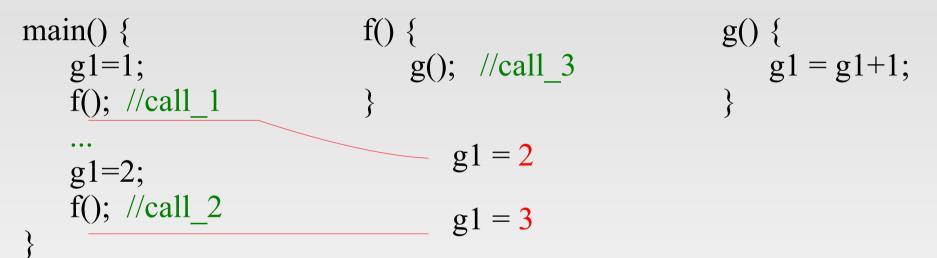
Interprocedural analysis

- W/out context, know nothing about g1 after calls
- *With* context, we can make statements about g1



- Background
- Benchmarks (0/7)
- Conclusion

Recall this example (without pointers) from class:



Interprocedural analysis

- W/out context, know nothing about g1 after calls
- With context, we can make statements about g1 at the cost of exponential code blowup or possibly infinite context strings

- Refresher
- Background
- Benchmarks (0/7)
- Conclusion

## **Points-to Analysis**

- Similar problem with pointers
  - What could p1 point to after a given call f()?
  - { $(pp \rightarrow p), (p \rightarrow x)?, (p \rightarrow y)?, (q \rightarrow z)$ }

- Refresher
- Background
- Benchmarks (0/7)
- Conclusion

# **Points-to Analysis**

- Similar problem with pointers
  - What could p1 point to after a given call f()?
  - $\{(pp \rightarrow p), (p \rightarrow x)?, (p \rightarrow y)?, (q \rightarrow z)\}$
- We would like to perform context-sensitive analysis, without the exponential blowup
  - Binary Decision Diagrams (BDDs) provide an efficient implementation, which we've seen

- Refresher
- Background
- Benchmarks (0/7)
- Conclusion

## **Points-to Analysis**

- Similar problem with pointers
  - What could p1 point to after a given call f()?
  - $\{(pp \rightarrow p), (p \rightarrow x)?, (p \rightarrow y)?, (q \rightarrow z)\}$
- We would like to perform context-sensitive analysis, without the exponential blowup
  - Binary Decision Diagrams (BDDs) provide an efficient implementation, which we've seen
- Efficient experimentation is now possible
  - Forms the basis of this paper

- Refresher
- Background
- Benchmarks (0/7)Conclusion

Examines the results of context-sensitive analysis

- Refresher
- Background
- Benchmarks (0/7)Conclusion

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively

- Refresher
- Background
- Benchmarks (0/7)
  Conclusion

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively
- Experiment with context abstraction models

- Refresher
- Background
- Benchmarks (0/7)Conclusion

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively
- Experiment with context abstraction models
  - Call-site sensitivity
  - Receiver object sensitivity
  - Zhu & Calman, Whaley & Lam (ZCWL) algorithm

- Refresher
- Background
- Benchmarks (0/7)
  Conclusion

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively
- Experiment with context abstraction models
  - Call-site sensitivity
  - Receiver object sensitivity
- 1, 2, and 3-level context strings
- 1H context-sensitive heap
- Zhu & Calman, Whaley & Lam (ZCWL) algorithm
  - Call-site abstraction. No bound on length of context string, but removes all cycles in context-insensitive graph to guarantee context string is finite.

- Refresher
- Background
- Benchmarks (0/7)
  Conclusion

## Benchmarks

	Total number of		Executed methods	
Benchmark	classes	methods	app.	+lib.
compress	41	476	56	463
db	32	440	51	483
jack	86	812	291	739
javac	209	2499	778	1283
jess	180	1482	395	846
mpegaudio	88	872	222	637
mtrt	55	574	182	616
soot-c	731	3962	1055	1549
sablecc-j	342	2309	1034	1856
polyglot	502	5785	2037	3093
antlr	203	3154	1099	1783
bloat	434	6125	138	1010
chart	1077	14966	854	2790
jython	270	4915	1004	1858
pmd	1546	14086	1817	2581
ps	202	1147	285	945

- Tests performed on several benchmark suites
  - SpecJVM 98, DaCapo v.beta050224, Ashes

\* from [1]

- Refresher
- Background
- Benchmarks (0/7)

#### Conclusion

	Total number of		Executed methods	
Benchmark	classes	methods	app.	+lib.
compress	41	476	56	463
db	32	440	51	483
jack	86	812	291	739
javac	209	2499	778	1283
jess	180	1482	395	846
mpegaudio	88	872	222	637
mtrt	55	574	182	616
soot-c	731	3962	1055	1549
sablecc-j	342	2309	1034	1856
polyglot	502	5785	2037	3093
antlr	203	3154	1099	1783
bloat	434	6125	138	1010
chart	1077	14966	854	2790
jython	270	4915	1004	1858
pmd	1546	14086	1817	2581
ps	202	1147	285	945

\* from [1]

#### **Benchmarks**

- Tests performed on several benchmark suites
  - SpecJVM 98, DaCapo v.beta050224, Ashes
- Context-insensitive baseline tested first
- All variations of objectsensitive, call-site, and ZCWL-based analyses compared against this reference

- Refresher
- Background
- Benchmarks (1/7)
  Conclusion

- Number of Contexts
- Simply a count of the number of contexts generated

- Refresher
- Background
- Benchmarks (1/7)
  Conclusion

- Number of Contexts
- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"

- Refresher
- Background
- Benchmarks (1/7)
   Conclusion

- Number of Contexts
- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"
- 1-level object-sensitive (OS) and call site (CS) contexts generate roughly 10-20 and 5-10 times the CI contexts
  - 1H sensitivity gives approximately the same numbers

- Refresher
- Background
- Benchmarks (1/7)
   Conclusion

- Number of Contexts
- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"
- 1-level object-sensitive (OS) and call site (CS) contexts generate roughly 10-20 and 5-10 times the CI contexts
  - 1H sensitivity gives approximately the same numbers
- 2-level sensitivity generates ~100-500 and ~125-350 times the CI contexts (3-level OS generates ~1500-25,000 times)

- Refresher
- Background
- Benchmarks (1/7)
   Conclusion

- Number of Contexts
- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"
- 1-level object-sensitive (OS) and call site (CS) contexts generate roughly 10-20 and 5-10 times the CI contexts
  - 1H sensitivity gives approximately the same numbers
- 2-level sensitivity generates ~100-500 and ~125-350 times the CI contexts (3-level OS generates ~1500-25,000 times)
- ZCWL generates between 2.9x10<sup>4</sup> and 2.1x10<sup>15</sup> times the contexts!

- Refresher
- Background
- Benchmarks (1/7)
- Conclusion

## Number of Contexts

- Huge numbers of contexts
  - Explicitly representing each one is a recipe for disaster

- Refresher
- Background
- Benchmarks (1/7)
  Conclusion

# Number of Contexts

- Huge numbers of contexts
  - Explicitly representing each one is a recipe for disaster
- Explains why previous analyses could not scale to the benchmarks used in this case study

- Refresher
- Background
- Benchmarks (2/7)
- Conclusion

Many contexts can be considered equivalent

- Refresher
- Background
- Benchmarks (2/7)
  Conclusion
- Equivalent Contexts
- Many contexts can be considered equivalent
- Formally:
  - Two method-context pairs (m<sub>1</sub>, c<sub>1</sub>) and (m<sub>2</sub>, c<sub>2</sub>) are equivalent if m<sub>1</sub> = m<sub>2</sub>, and any local pointer p has the same points-to set in both contexts

- Refresher
- Background
- Benchmarks (2/7)
  Conclusion

- **Equivalent Contexts**
- Many contexts can be considered equivalent
- Formally:
  - Two method-context pairs (m<sub>1</sub>, c<sub>1</sub>) and (m<sub>2</sub>, c<sub>2</sub>) are equivalent if m<sub>1</sub> = m<sub>2</sub>, and any local pointer p has the same points-to set in both contexts
- If there are many equivalent contexts in an analysis, explicitly storing each one separately is a waste

- Refresher
- Background
- Benchmarks (2/7)
  Conclusion

- **Equivalent Contexts**
- Many contexts can be considered equivalent
- Formally:
  - Two method-context pairs (m<sub>1</sub>, c<sub>1</sub>) and (m<sub>2</sub>, c<sub>2</sub>) are equivalent if m<sub>1</sub> = m<sub>2</sub>, and any local pointer p has the same points-to set in both contexts
- If there are many equivalent contexts in an analysis, explicitly storing each one separately is a waste
- However, methods to determine equivalent contexts prior to analysis have yet to be discovered

- Refresher
- Background
- Benchmarks (2/7)
- Conclusion

 In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted

- Refresher
- Background
- Benchmarks (2/7)
- Conclusion

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously

- Refresher
- Background
- Benchmarks (2/7)
- Conclusion

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously
- Maximum of 33.8 times the CI contexts in the case of 3-level OS analysis (from 13,289, previously)

- Refresher
- Background
- Benchmarks (2/7)
- Conclusion

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously
- Maximum of 33.8 times the CI contexts in the case of 3-level OS analysis (from 13,289, previously)
- ZCWL showed the greatest improvement:

- Refresher
- Background
- Benchmarks (2/7)
- Conclusion

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously
- Maximum of 33.8 times the CI contexts in the case of 3-level OS analysis (from 13,289, previously)
- ZCWL showed the greatest improvement: from between 2.9x10<sup>4</sup> and 2.1x10<sup>15</sup> times to only ~3-7

- Refresher
- Background
- Benchmarks (2/7)
- Conclusion

# **Equivalent Contexts**

 Finding equivalent contexts a priori would clearly benefit analysis

- Refresher
- Background
- Benchmarks (2/7)
  Conclusion

- **Equivalent Contexts**
- Finding equivalent contexts a priori would clearly benefit analysis
- Notes:
  - OS-based analysis generated (~3x) more equivalent contexts, which would likely make it more precise than CSs
  - Longer context strings led to an exponential increase in space required, but only minimal precision improvements
  - ZCWL models cycles insensitively; thus drastically reducing the number of equivalent contexts generated

- Refresher
- Background
- Benchmarks (3/7)
- Conclusion

## **Distinct Points-to sets**

Can be seen as a rough approximation of the spaceefficiency of BDDs

- Refresher
- Background
- Benchmarks (3/7)
  Conclusion

- **Distinct Points-to sets**
- Can be seen as a rough approximation of the spaceefficiency of BDDs
- In nearly all types of context abstraction (OS-, CS-, and ZCWL-based), there is no significant advantage over CI analysis

- Refresher
- Background
- Benchmarks (3/7)
  Conclusion

- **Distinct Points-to sets**
- Can be seen as a rough approximation of the spaceefficiency of BDDs
- In nearly all types of context abstraction (OS-, CS-, and ZCWL-based), there is no significant advantage over CI analysis
  - However, 1-level context sensitive heap abstractions led to an 11-fold increase.
  - Points-to sets are pairs of abstract objects and contexts, rather than simply the objects themselves

- Refresher
- Background
- Benchmarks (3/7)
  Conclusion

- **Distinct Points-to sets**
- Can be seen as a rough approximation of the spaceefficiency of BDDs
- In nearly all types of context abstraction (OS-, CS-, and ZCWL-based), there is no significant advantage over CI analysis
  - However, 1-level context sensitive heap abstractions led to an 11-fold increase.
  - Points-to sets are pairs of abstract objects and contexts, rather than simply the objects themselves
- Representing points-to sets less critical than efficiently representing contexts

- Refresher
- Background
- Benchmarks (4/7)
- Conclusion

 Reducing the number of reachable methods allows dead-code elimination

- Refresher
- Background
- Benchmarks (4/7)
- Conclusion

- Reducing the number of reachable methods allows dead-code elimination
- Context-sensitive graphs were created, then context "projected" away to enable comparison

- Refresher
- Background
- Benchmarks (4/7)
- Conclusion

- Reducing the number of reachable methods allows dead-code elimination
- Context-sensitive graphs were created, then context "projected" away to enable comparison
- Results underwhelming: maximum of 13 methods fewer than CI approach

- Refresher
- Background
- Benchmarks (4/7)
- Conclusion

- Reducing the number of reachable methods allows dead-code elimination
- Context-sensitive graphs were created, then context "projected" away to enable comparison
- Results underwhelming: maximum of 13 methods fewer than CI approach
- Results were slightly better for OS-based analysis
  - Node-visitor algorithms where certain types of nodes will never be reached
  - Heap abstractions improve performance on dynamicallyallocated objects

- Refresher
- Background
- Benchmarks (5/7)
- Conclusion



- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable

- Refresher
- Background
- Benchmarks (5/7)
- Conclusion



- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...

- Refresher
- Background
- Benchmarks (5/7)
- Conclusion



- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...
- ...except in sablecc-j benchmark w/ 1-level OS

- Refresher
- Background
- Benchmarks (5/7)
- Conclusion



- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...
- ...except in sablecc-j benchmark w/ 1-level OS
- Benchmark uses tree traversal with numerous this.getParent() calls. W/out context, this could generate a huge number of potential call edges

- Refresher
- Background
- Benchmarks (5/7)
- Conclusion



- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...
- ...except in sablecc-j benchmark w/ 1-level OS
- Benchmark uses tree traversal with numerous this.getParent() calls. W/out context, this could generate a huge number of potential call edges
  - 17,925 call edges in CI analysis, only ~5100 in context sensitive test

- Refresher
- Background
- Benchmarks (6/7)
- Conclusion

 At compile time, virtual calls introduce potential call edges between pointers and any number of targets

- Refresher
- Background
- Benchmarks (6/7)
  Conclusion

- Virtual Call Resolution
- At compile time, virtual calls introduce potential call edges between pointers and any number of targets
- Reducing potential polymorphism of call sites reduces the amount of call edges generated
  - In effect, a subset of the call-edge problem, previously

- Refresher
- Background
- Benchmarks (6/7)
  Conclusion

- Virtual Call Resolution
- At compile time, virtual calls introduce potential call edges between pointers and any number of targets
- Reducing potential polymorphism of call sites reduces the amount of call edges generated
  - In effect, a subset of the call-edge problem, previously
- Fully resolving a call site (i.e., removing polymorphism) means it can be replaced by cheaper static methods, allowing further optimization

- Refresher
- Background
- Benchmarks (6/7)
- Conclusion

Again, relatively small improvements

- Refresher
- Background
- Benchmarks (6/7)
  Conclusion

- Again, relatively small improvements
- CS-based optimization performs as well as, but never better, than OS-based

- Refresher
- Background
- Benchmarks (6/7)
  Conclusion

- Again, relatively small improvements
- CS-based optimization performs as well as, but never better, than OS-based
- Once again, sablecc-j provides a good example
  - Some devirtualization can be handled by any contextsensitive analysis
  - A further set of devirtualization requires OS
  - A final set requires context-sensitive heap objects

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion



 In an OO language, a cast cannot fail if the pointer that it is casting can only point to variables that are subtypes of the cast

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion



- In an OO language, a cast cannot fail if the pointer that it is casting can only point to variables that are subtypes of the cast
  - Presumably, proving that certain casts cannot fail reduces the number of exceptional call edges (Comments?)

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion



 Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion



- Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions
- In polyglot benchmark, the number of potentially failing casts is reduced from 3539 (CI) to 1017.

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion



- Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions
- In polyglot benchmark, the number of potentially failing casts is reduced from 3539 (CI) to 1017.
- This benchmark involves a large class hierarchy, in which each subclass implements a copy() method
- Using OS, receiver objects performing the casts can be determined, and cast safety made more precise

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion



- Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions
- In polyglot benchmark, the number of potentially failing casts is reduced from 3539 (CI) to 1017.
- This benchmark involves a large class hierarchy, in which each subclass implements a copy() method
- Using OS, receiver objects performing the casts can be determined, and cast safety made more precise
- Further, OS heap abstractions can more accurately model casts in dynamically-allocated objects

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, and now we have seen how

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, and now we have seen how
- Generally, context-sensitive points-to analysis:
  - improved call-graph precision slightly
  - improved virtual call resolution even more
  - led to major precision improvements in cast safety analysis

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, and now we have seen how
- Generally, context-sensitive points-to analysis:
  - improved call-graph precision slightly
  - improved virtual call resolution even more
  - led to major precision improvements in cast safety analysis
- OS-based approaches were never less precise than CS-based ones, and scaled better than the latter when context string length was increased

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, and now we have seen how
- Generally, context-sensitive points-to analysis:
  - improved call-graph precision slightly
  - improved virtual call resolution even more
  - led to major precision improvements in cast safety analysis
- OS-based approaches were never less precise than CS-based ones, and scaled better than the latter when context string length was increased
- The ZCWL algorithm was never more precise than OS

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion

 Extending the context strings for OS-based analysis gave only modest performance increases, but contextsensitive heaps gave, in the best case, significant improvements

- Refresher
- Background
- Benchmarks (7/7)
- Conclusion

- Extending the context strings for OS-based analysis gave only modest performance increases, but contextsensitive heaps gave, in the best case, significant improvements
- However, efficiently implementing 1H-object-sensitive analysis without BDDs requires further work

- Refresher
- Background
- Benchmarks
- Conclusion



- Discussion of ZCWL algorithm and context-sensitive heaps would benefit the reader
- Are benchmarks particularly suited to OS-based analysis? Are there no benchmarks for which a CSbased approach would show greater improvement?

