Alias Analysis for Object-Oriented Programs

M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav

Paper Review

Adilet Zhaxybay

Nazarbayev University

Outline

Outline

Outline

Alias Analysis

Aliases

Two pointers said to be *aliases* if they point to the same location

Alias Analysis

Aliases

Two pointers said to be *aliases* if they point to the same location

Alias Analysis

Analysis which determines which pointers may or must be aliases

Example of Aliases

Example in C

```
int * x, y, z;
x = malloc(sizeof(int));
y = x; // x and y are aliases now
z = malloc(sizeof(int)); // x and z are not aliases
```

Paper Structure

Origin

Work by Sridharan et al. gives a high-level survey of the alias-analiysis techniques that authors have found most-useful during a years-long effort developing industrial-strength analyses for Java programs.

Paper Structure

Origin

Work by Sridharan et al. gives a high-level survey of the alias-analiysis techniques that authors have found most-useful during a years-long effort developing industrial-strength analyses for Java programs.

Published in 2013 in ‘Lecture Notes in Computer Science‘

Paper Structure

Origin

Work by Sridharan et al. gives a high-level survey of the alias-analiysis techniques that authors have found most-useful during a years-long effort developing industrial-strength analyses for Java programs.

Published in 2013 in ‘Lecture Notes in Computer Science‘

It is *not* an exhaustive survey of alias analysis.

Paper Structure

Origin

Work by Sridharan et al. gives a high-level survey of the alias-analiysis techniques that authors have found most-useful during a years-long effort developing industrial-strength analyses for Java programs.

Published in 2013 in ‘Lecture Notes in Computer Science‘

It is *not* an exhaustive survey of alias analysis.

Challenges

Treats alias analysis as a constrant tradeoff between *scalability* (adaptation to large programs) and *precision* (accuracy of analysis).

Paper Focus

Focus

Paper focuses on two main techniques:

Paper Focus

Focus

Paper focuses on two main techniques:

Points-to analysis — analysis, that can be used to determine *may-alias* information, i.e., whether it is possible for two pointers to be aliased during program execution.

Paper Focus

Focus

Paper focuses on two main techniques:

Points-to analysis — analysis, that can be used to determine *may-alias* information, i.e., whether it is possible for two pointers to be aliased during program execution.

Access-path tracking — provides *must-alias* information, i.e., whether two pointers must be aliased at some program point.

Paper Focus

Focus

Paper focuses on two main techniques:

Points-to analysis — analysis, that can be used to determine *may-alias* information, i.e., whether it is possible for two pointers to be aliased during program execution.

Access-path tracking — provides *must-alias* information, i.e., whether two pointers must be aliased at some program point.

Java

Additionally paper aims to explain particlar challenges for modern Java programs.

Outline

Possible Errors

Memory leak

```
int * x = malloc(sizeof(int));
x = malloc(sizeof(int));
```

Possible Errors

Memory leak

```
int * x = malloc(sizeof(int));
x = malloc(sizeof(int));
```

Invalid memory access

```
int * x = malloc(sizeof(int));
int * y = x;
free x;
free y;
```

More Sophisticated Example

```
1  public void test(File file, String enc) throws IOException {
2    PrintWriter out = null;
3    try {
4      try {
5        out = new PrintWriter(
6                new OutputStreamWriter(
7                    new FileOutputStream(file), enc));
8      } catch (UnsupportedEncodingException ue) {
9        out = new PrintWriter(new FileWriter(file));
10     }
11     out.append('c');
12   } catch (IOException e) {
13   } finally {
14     if (out != null) {
15       out.close();
16     }
17   }
18 }
```

Two Alias Analysis Characteristics

Flow-sensitivity

Flow-sensitive analysis computes aliases for all flow paths in the program, while flow-insensitive analysis computes aliasing for the program as a whole.

Two Alias Analysis Characteristics

Flow-sensitivity

Flow-sensitive analysis computes aliases for all flow paths in the program, while flow-insensitive analysis computes aliasing for the program as a whole.

Context-sensitivity

Context-sensitivity is about function/procedure calls and means whether a context of a call is taken into consideration or not.

Outline

Outline

Point-to Analysis

Paper presents several common variants of Andersen's point-to analysis.

Point-to Analysis

Paper presents several common variants of Andersen's point-to analysis.

Point-to analysis

A points-to analysis computes an over-approximation of the heap locations that each program pointer may point to. Pointers include program variables and also pointers within heap-allocated objects, e.g., instance fields.

Point-to Analysis

Paper presents several common variants of Andersen's point-to analysis.

Point-to analysis

A points-to analysis computes an over-approximation of the heap locations that each program pointer may point to. Pointers include program variables and also pointers within heap-allocated objects, e.g., instance fields.

The result of the analysis is a points-to relation pt, with pt(p) representing the points-to set of a pointer p.

Point-to Analysis

Paper presents several common variants of Andersen's point-to analysis.

Point-to analysis

A points-to analysis computes an over-approximation of the heap locations that each program pointer may point to. Pointers include program variables and also pointers within heap-allocated objects, e.g., instance fields.

The result of the analysis is a points-to relation pt, with pt(p) representing the points-to set of a pointer p.

Point-to analysis is flow insensitive: it assumes statements can execute in any order and any number of times.

Point-to Analysis Statements

Statement	Constraint	
`i: x = new T()`	$\{o_i\} \subseteq pt(x)$	[New]
`x = y`	$pt(y) \subseteq pt(x)$	[Assign]
`x = y.f`	$\dfrac{o_i \in pt(y)}{pt(o_i.\mathtt{f}) \subseteq pt(x)}$	[Load]
`x.f = y`	$\dfrac{o_i \in pt(x)}{pt(y) \subseteq pt(o_i.\mathtt{f})}$	[Store]

Table 1. Canonical statements for context-insensitive Java points-to analysis and the corresponding points-to set constraints.

Context Sensitivity

Context sensitivity

It is possible to extend point-to analysis to incorporate context-sensitive handling of method calls.

Context Sensitivity

Context sensitivity

It is possible to extend point-to analysis to incorporate context-sensitive handling of method calls.

A context-sensitive points-to analysis separately analyzes a method m for each calling context that arises at call sites of m.

Context Sensitivity

Context sensitivity

It is possible to extend point-to analysis to incorporate context-sensitive handling of method calls.

A context-sensitive points-to analysis separately analyzes a method m for each calling context that arises at call sites of m.

Separately analyzing a method for each context removes imprecision due to the merge of analysis results across its invocations.

Context Sensitivity

Context sensitivity

It is possible to extend point-to analysis to incorporate context-sensitive handling of method calls.

A context-sensitive points-to analysis separately analyzes a method m for each calling context that arises at call sites of m.

Separately analyzing a method for each context removes imprecision due to the merge of analysis results across its invocations.

```
1 id(p) { return p; }
2 x = new Object(); // o1
3 y = new Object(); // o2
4 a = id(x);
5 b = id(y);
```

Point-to Analysis Context-sensitive Statements

Statement in method m	Constraint	
`i: x = new T()`	$\dfrac{c \in contexts(m)}{\langle o_i, heapSelector(c)\rangle \in pt(\langle x, c\rangle)}$	[New]
`x = y`	$\dfrac{c \in contexts(m)}{pt(\langle y, c\rangle) \subseteq pt(\langle x, c\rangle)}$	[Assign]
`x = y.f`	$\dfrac{c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle y, c\rangle)}{pt(\langle o_i, c'\rangle.\mathtt{f}) \subseteq pt(\langle x, c\rangle)}$	[Load]
`x.f = y`	$\dfrac{c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle x, c\rangle)}{pt(\langle y, c\rangle) \subseteq pt(\langle o_i, c'\rangle.\mathtt{f})}$	[Store]
`j: x = r.g(a_1,...,a_n)`	$\dfrac{\begin{array}{c} c \in contexts(m) \qquad \langle o_i, c'\rangle \in pt(\langle r, c\rangle) \\ m' = dispatch(\langle o_i, c'\rangle, \mathtt{g}) \\ argvals = [\{\langle o_i, c'\rangle\}, pt(\langle a_1, c\rangle), \ldots, pt(\langle a_n, c\rangle)] \\ c'' \in selector(m', c, j, argvals) \end{array}}{\begin{array}{c} c'' \in contexts(m') \\ \langle o_i, c'\rangle \in pt(\langle m'_{this}, c''\rangle) \\ pt(\langle a_k, c\rangle) \subseteq pt(\langle m'_{p_k}, c''\rangle),\ 1 \le k \le n \\ pt(\langle m'_{ret}, c''\rangle) \subseteq pt(\langle x, c\rangle) \end{array}}$	[Invoke]
`return x`	$\dfrac{c \in contexts(m)}{pt(\langle x, c\rangle) \subseteq pt(\langle m_{ret}, c\rangle)}$	[Return]

Table 2. Inference rules for context-sensitive points-to analysis.

Context Sensitivity Variants

Call strings

A standard technique to distinguish contexts is via *call strings*, which abstract the possible call stacks under which a method may be invoked. Call strings are typically represented as a sequence of call site identifiers, corresponding to a (partial) call stack.

Context Sensitivity Variants

Call strings

A standard technique to distinguish contexts is via *call strings*, which abstract the possible call stacks under which a method may be invoked. Call strings are typically represented as a sequence of call site identifiers, corresponding to a (partial) call stack.

Call string are often k-limited (only first k call site identifiers are considered).

Context Sensitivity Variants

Call strings

A standard technique to distinguish contexts is via *call strings*, which abstract the possible call stacks under which a method may be invoked. Call strings are typically represented as a sequence of call site identifiers, corresponding to a (partial) call stack.

Call string are often k-limited (only first k call site identifiers are considered).

Object sensitivity

Technique to distinguish contexts via objects which invoke methods.

Context Sensitivity Variants

Call strings

A standard technique to distinguish contexts is via *call strings*, which abstract the possible call stacks under which a method may be invoked. Call strings are typically represented as a sequence of call site identifiers, corresponding to a (partial) call stack.

Call string are often k-limited (only first k call site identifiers are considered).

Object sensitivity

Technique to distinguish contexts via objects which invoke methods.

And there many more approaches and variants to distinguish contexts.

Outline

Algorithm

Algorithm

Algorithm, implementing Andersen's context-insensitive ponts-to analysis, constructs a flow graph G representing the pointer flow for a program.

Algorithm

Algorithm

Algorithm, implementing Andersen's context-insensitive ponts-to analysis, constructs a flow graph G representing the pointer flow for a program.

G has nodes for variables, abstract locations, and field of abstract locations.

Algorithm

Algorithm

Algorithm, implementing Andersen's context-insensitive ponts-to analysis, constructs a flow graph G representing the pointer flow for a program.

G has nodes for variables, abstract locations, and field of abstract locations.

G has an edge $n \rightarrow n'$ iff one of the following two conditions holds:

1. n is an abstract location o_i representing a statement x = new T(), and n' is x.
2. $pt(n) \subseteq pt(n')$ according to some rule.

Algorithm Pseudocode

```
DoAnalysis()
 1  for each statement i: x = new T() do
 2          pt_Δ(x) ← pt_Δ(x) ∪ {o_i}, o_i fresh
 3          add x to worklist
 4  for each statement x = y do
 5          add edge y → x to G
 6  while worklist ≠ ∅ do
 7          remove n from worklist
 8          for each edge n → n' ∈ G do
 9                  DiffProp(pt_Δ(n), n')
10          if n represents a local x
11            then for each statement x.f = y do
12                          for each o_i ∈ pt_Δ(n) do
13                                  if y → o_i.f ∉ G
14                                    then add edge y → o_i.f to G
15                                         DiffProp(pt(y), o_i.f)
16                  for each statement y = x.f do
17                          for each o_i ∈ pt_Δ(n) do
18                                  if o_i.f → y ∉ G
19                                    then add edge o_i.f → y to G
20                                         DiffProp(pt(o_i.f), y)
21          pt(n) ← pt(n) ∪ pt_Δ(n)
22          pt_Δ(n) ← ∅

DiffProp(srcSet, n)
1  pt_Δ(n) ← pt_Δ(n) ∪ (srcSet − pt(n))
2  if pt_Δ(n) changed then add n to worklist
```

Complexity

Complexity

Algorithm from the previous slide has worst-case cubic complexity.

Complexity

Complexity

Algorithm from the previous slide has worst-case cubic complexity.

In practice, many papers have reported scaling behavior significantly better than cubic (papers reporting analysis of millions of lines of code are now commonplace).

Optimizations

Type filters

Algorithm from the previous slide has worst-case cubic complexity.In strongly-typed languages, type filters provide a simple but highly effective optimization which improves both precision and (usually) performance.

Optimizations

Type filters

Algorithm from the previous slide has worst-case cubic complexity.In strongly-typed languages, type filters provide a simple but highly effective optimization which improves both precision and (usually) performance.

Cycle elimination

Collapse cycles in the flow graph.

Optimizations

Type filters

Algorithm from the previous slide has worst-case cubic complexity.In strongly-typed languages, type filters provide a simple but highly effective optimization which improves both precision and (usually) performance.

Cycle elimination

Collapse cycles in the flow graph.

Method-local state

If a variable's points-to set is determined entirely by statements in the enclosing method, the points-to set is computed on-demand rather than via the global constraint system.

Other Directions

Context sensitivity

The most straightforward strategy for implementing context sensitivity is via *cloning*.

Other Directions

Context sensitivity

The most straightforward strategy for implementing context sensitivity is via *cloning*.

There are also many other strategies and solutions usually relying on clever data structures.

Other Directions

Context sensitivity

The most straightforward strategy for implementing context sensitivity is via *cloning*.

There are also many other strategies and solutions usually relying on clever data structures.

Demand-driven analysis

The previous discussion focused on computing an *exhaustive* points-to analysis solution, i.e., computing all points-to sets for a program.

Other Directions

Context sensitivity

The most straightforward strategy for implementing context sensitivity is via *cloning*.

There are also many other strategies and solutions usually relying on clever data structures.

Demand-driven analysis

The previous discussion focused on computing an *exhaustive* points-to analysis solution, i.e., computing all points-to sets for a program.

However, this is usually not required in practice.

Outline

Must-Alias Analysis

Must-alias analysis

Paper presents a flow-sensitive must-alias analysis based on *access paths*

Must-Alias Analysis

Must-alias analysis

Paper presents a flow-sensitive must-alias analysis based on *access paths*

Must-alias analysis provides information about whether two pointers must be aliased at some program point.

Must-Alias Analysis Importance

```
1  File makeFile {
2    return new File(); //  ⟨o1, init⟩, ⟨o2, init⟩
3  }
4  File f = makeFile(); //  ⟨o1, init⟩, ⟨o2, init⟩
5  File g = makeFile(); //  ⟨o1, init⟩, ⟨o2, init⟩
6  if(?)
7    f.open(); //  ⟨o1, open⟩, ⟨o2, init⟩
8  else
9    g.open();
10 f.read(); //  ⟨o1, open⟩, ⟨o2, init⟩
11 g.read(); //  ⟨o1, open⟩, ⟨o2, err⟩
12 }
```

Fig. 4. Concrete states for a program reading from two `File` objects allocated at the same allocation site. The example shows states for an execution in which the condition evaluates to true.

May-Alias Analysis Drawback

```
1  File makeFile {
2    return new File(); //  ⟨A, init⟩
3  }
4  File f = makeFile(); //  ⟨A, init⟩
5  File g = makeFile(); //  ⟨A, init⟩
6  if(?)
7    f.open(); //  ⟨A, open⟩
8  else
9    g.open(); //  ⟨A, open⟩
10 f.read(); //  ⟨A, open⟩
11 g.read(); //  ⟨A, open⟩
12 }
```

Fig. 5. Unsound update of abstract states for the example of Fig. 4.

Weak Updates

Weak update

Weak update reflects all possible states of an object.
However, this fails even in simple cases.

```
1 File f = new File(); //  ⟨A, init⟩
2 f.open(); //  ⟨A, init⟩, ⟨A, open⟩
3 f.read(); //  ⟨A, err⟩, ⟨A, open⟩
```

Fig. 6. Simple correct example that cannot be verified directly using weak updates.

Must-alias information

Correct analysis of this requires must-alias information

Access Paths

Access path

Must-alias information is maintained with a help of *access paths* — a sequences of references that point to a heap allocated object.

Maintaining Must Points-to Information

To describe the abstraction, first assume that a preliminary flow-insensitive points-to analysis has run.

Maintaining Must Points-to Information

To describe the abstraction, first assume that a preliminary flow-insensitive points-to analysis has run.

Aliasing relationships in the form of tuples

We represent aliasing relationships with tuples of the form $\langle o, unique, AP_m, May, AP_{mn} \rangle$ where:
- o is an instance key (abstract memory location from the preliminary points-to analysis).
- *unique* indicates whether the corresponding allocation site has a single concrete live object.
- AP_m is a set of access paths that must point-to o.
- *May* is a boolean that indicates whether there are access paths (not in the must set) that may point to o.
- AP_{mn} is a set of access paths that do not point-to o.

Tuple Transformers

Stmt S	Resulting abstract tuples
$v = \texttt{new T()}$ where $o =$ Stmt S	$\langle o, false, AP_{\text{M}} \setminus startsWith(v, AP_{\text{M}}), May, AP_{\text{MN}} \cup \{v\}\rangle$ $\langle o, true, \{v\}, false, \emptyset\rangle$
$v = \texttt{null}$	$\langle o, unique, AP'_{\text{M}}, May, AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \setminus startsWith(v, AP_{\text{M}})$ $AP_{\text{MN}} := AP_{\text{MN}} \cup \{v\}$
$v.f = \texttt{null}$	$\langle o, unique, AP'_{\text{M}}, May, AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \setminus \{e'.f.\gamma \mid mayAlias(e', v), \gamma \in \Gamma\}$ $AP'_{\text{MN}} := AP_{\text{MN}} \cup \{v.f\}$
$v = e$	$\langle o, unique, AP'_{\text{M}}, May, AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \cup \{v.\gamma \mid e.\gamma \in AP_{\text{M}}\}$ $AP'_{\text{MN}} := AP_{\text{MN}} \setminus \{v \mid e \notin AP_{\text{MN}}\}$
$v.f = e$	$\langle o, unique, AP'_{\text{M}}, May', AP'_{\text{MN}}\rangle$ $AP'_{\text{M}} := AP_{\text{M}} \cup \{v.f.\gamma \mid e.\gamma \in AP_{\text{M}}\}$ $May' := May \vee \exists v.f.\gamma \in AP'_{\text{M}}.\exists p \in AP.mayAlias(v, p) \wedge p.f.\gamma \notin AP'_{\text{M}}$ $AP'_{\text{MN}} := AP_{\text{MN}} \setminus \{v.f \mid e \notin AP_{\text{MN}}\}$
$startsWith(v, P) = \{v.\gamma \mid \gamma \in P\}$	

Table 3. Transfer functions for statements indicating how an incoming tuple $\langle o, unique, AP_{\text{M}}, May, AP_{\text{MN}}\rangle$ is transformed, where $pt(e)$ is the set of instance keys pointed-to by e in the flow-insensitive solution, $v \in VarId$. $mayAlias(e_1, e_2)$ iff pointer analysis indicates e_1 and e_2 may point to the same instance key.

Strong Updates with Access Paths

```
1 Collection files = ...
2 while (...) {
3  File f = new File(); // (⟨A, true, {f}, false, ∅⟩, init), (⟨A, false, ∅, true, {f}⟩, open)
4  files.add(f); // (⟨A, true, {f}, true, ∅⟩, init), (⟨A, false, ∅, true, {f}⟩, open)
5  f.open(); // (⟨A, true, {f}, true, ∅⟩, open), (⟨A, false, ∅, true, {f}⟩, open)
6  f.read(); // (⟨A, true, {f}, true, ∅⟩, open), (⟨A, false, ∅, true, {f}⟩, open)
7 }
```

Fig. 7. Illustration of a strong update to the state of a `File` object using access paths.

Outline

Points-to Analysis Difficulties

Reflection

In Java-like languages, reflection allows for meta-programming based on string names of program constructs like classes or methods.

Points-to Analysis Difficulties

Reflection

In Java-like languages, reflection allows for meta-programming based on string names of program constructs like classes or methods.

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

Points-to Analysis Difficulties

Reflection

In Java-like languages, reflection allows for meta-programming based on string names of program constructs like classes or methods.

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

Analyzing this code with the assumption that x may name any type yields extremely imprecise results.

Points-to Analysis Difficulties

Reflection

In Java-like languages, reflection allows for meta-programming based on string names of program constructs like classes or methods.

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

Analyzing this code with the assumption that x may name any type yields extremely imprecise results.

Libraries and frameworks

Also, usage of large number of libraries and frameworks (with many of them using reflection) makes analysis much more difficult.

Under-Approximate Techniques

Under-approximate techniques

Possible solution of problems of Java-like languages.

Under-Approximate Techniques

Under-approximate techniques

Possible solution of problems of Java-like languages.

Uses simpler versions of access path method, or some other domain-specific techniques. In general produces less accurate and sound result.

Outline

Conclusion

Conclusion

Paper have presented a high-level overview of state-of-the-art may- and must-alias analyses for object-oriented programs, based on authors' experiences implementing production-quality static analyses for Java. The sound alias-analysis techniques presented there work well for medium-sized programs, while for large-scale Java programs, an under-approximate alias analysis based on access-path tracking currently yields the most useful results.

Future Work

Potentially fruitful directions for future work:

Future Work

Potentially fruitful directions for future work:

Reflection

Solution of reflection problem for Java-like languages

Future Work

Potentially fruitful directions for future work:

Reflection

Solution of reflection problem for Java-like languages

Dynamically-Typed Languages

Analysis of languages like JavaScript.

Future Work

Potentially fruitful directions for future work:

Reflection

Solution of reflection problem for Java-like languages

Dynamically-Typed Languages

Analysis of languages like JavaScript.

Developer Tool Integration

Integrate alias analysis with some developer tools.

Thank you for attention

Questions?

References

1. Alias analysis for object-oriented programs (by M. Sridharan, S. Chandra, J. Dolby, S.J. Fink, and E. Yahav, Lecture Notes in Computer Science, 2013, Vol. 7850, p. 196-232.)
2. Alias Calculus for a Simple Imperative Language with Decidable Pointer Arithmetic (Nikolay Shilov, Aizhan Satekbayeva, Aleksandr P. Vorontsov)