Exception Analysis and Points-to Analysis

Better Together

Martin Bravenboer

LogicBlox

Yannis Smaragdakis

UMass Amherst

ISSTA 2009

International Symposium on Software Testing and Analysis

what do we do?

precise analysis of exception handling
improve precision and speed of points-to analyses

what do we do?

- precise analysis of exception handling
- improve precision and speed of points-to analyses

how do we do it?

- fully declarative specification
- modular extension

what do we do?

precise analysis of exception handling
improve precision and speed of points-to analyses

how do we do it?

fully declarative specification
modular extension

why do you care?

fast, sophisticated, simple
different

DOOP

what do we do?

precise analysis of exception handling
improve precision and speed of points-to analyses

how do we do it?

fully declarative specification
modular extension

why do you care?

fast, sophisticated, simple
different

why is it relevant?

major new experimental findings
state-of-the-art points-to analyses

DOOP

computation of control-flow induced by exceptions

```
void foo() {
  if(...)
    throw new FooException();
}

void mid() {
  foo();
}

void bar() {
  try {
    foo();
  }
  catch(Exception exc) {...}
}
```

computation of control-flow induced by exceptions

```
void foo() {
  if(...)
    throw new FooException();
}

void mid() {
  foo();
}

void bar() {
  try {
    foo();
  }
  catch(Exception exc) {...}
}
```

- exception-flow induces interprocedural assignments
- exceptions are normal objects
- arbitrary expressions can be thrown

computation of control-flow induced by exceptions

```
void foo() {
  if(...)
    throw new FooException();
}

void mid() {
  foo();
}

void bar() {
  try {
    foo();
  }
  catch(Exception exc) {...}
}
```

- exception-flow induces interprocedural assignments
- exceptions are normal objects
- arbitrary expressions can be thrown

```
catch(SomeException e) {
  throw e.getCause();
}
```

```
throw
  createSomeException();
```

what is exception analysis?

computation of control-flow induced by exceptions

```
void foo() {
  if(...)
    throw new FooException();
}

void mid() {
  foo();
}

void bar() {
  try {
    foo();
  }
  catch(Exception exc) {...}
}
```

questions answered:

- what exceptions may `foo` throw?
- where may the `FooException` thrown in `foo` get caught?
- what exceptions may get caught by the handler in `bar`?

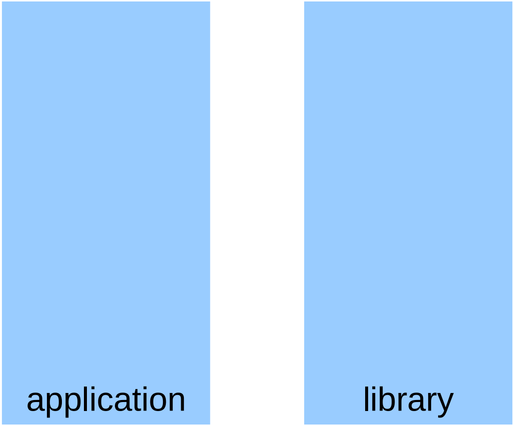
application: program understanding

- understand exception-flow in codebases
- coding assistance tool
- also for languages with declared checked exceptions
  - unchecked exceptions
  - `throws`-clause specifies superset
  - e.g. `IOException`

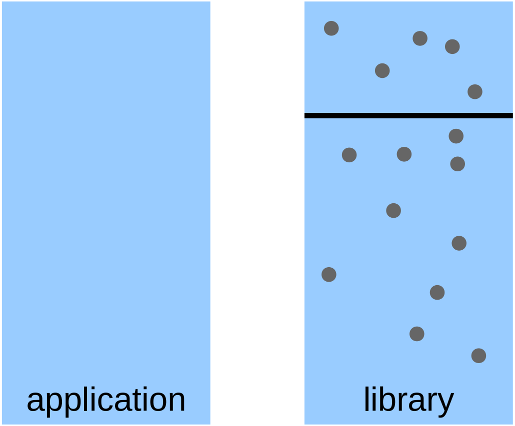
⇒ exception types do not explain where exceptions originate

why exception analysis? (2)

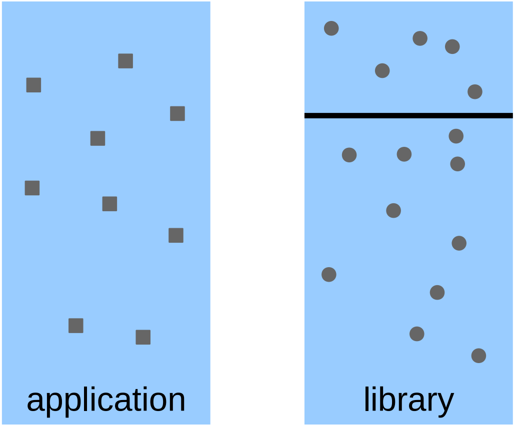
application: test coverage of exceptional situations [Fu et al.]



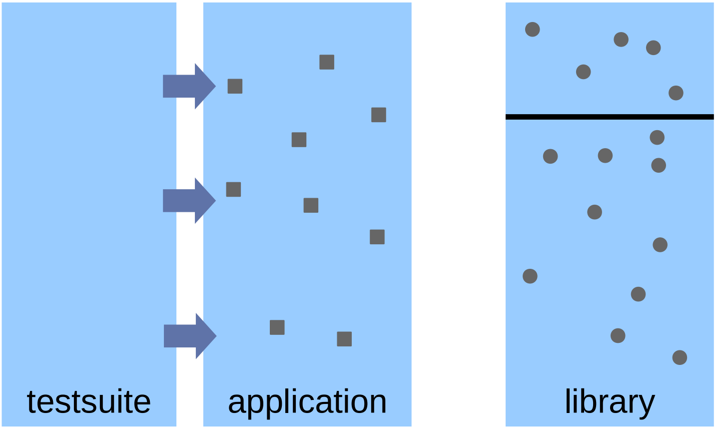
application: test coverage of exceptional situations [Fu et al.]



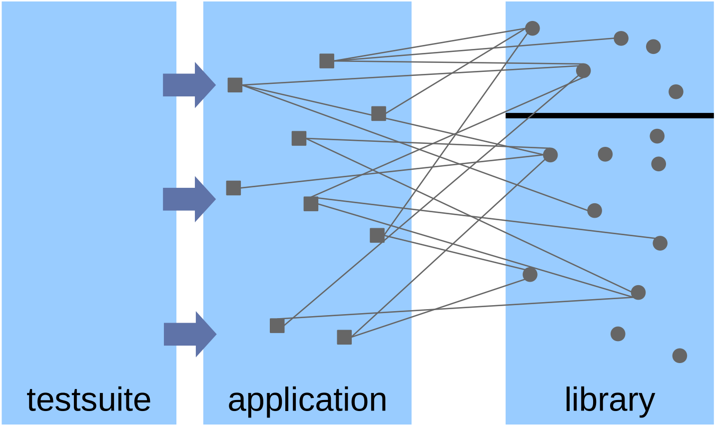
application: test coverage of exceptional situations [Fu et al.]



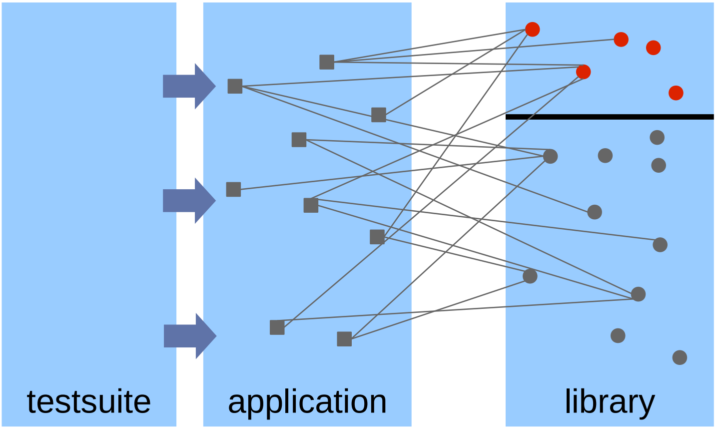
application: test coverage of exceptional situations [Fu et al.]



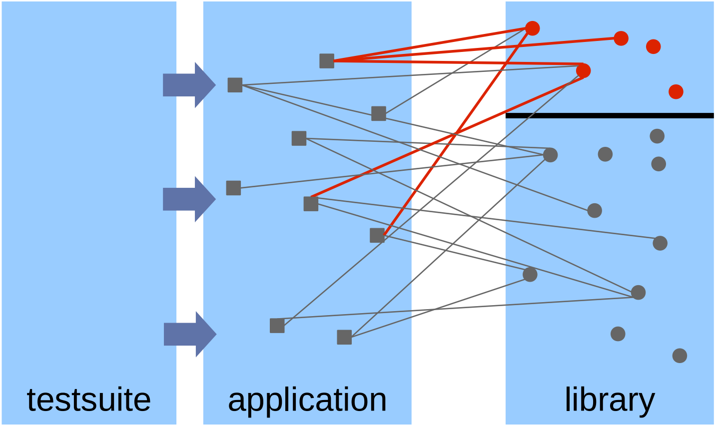
application: test coverage of exceptional situations [Fu et al.]



application: test coverage of exceptional situations [Fu et al.]



application: test coverage of exceptional situations [Fu et al.]



why exception analysis? (3)

points-to analysis (facilitate other applications)

what objects can a variable point to?

program

```
void foo() {
  a = new A1();
  b = id(a);
}

void bar() {
  a = new A2();
  b = id(a);
}

A id(A a) {
  return a;
}
```

what objects can a variable point to?

program

```
void foo() {
  a = new A1();
  b = id(a);
}

void bar() {
  a = new A2();
  b = id(a);
}

A id(A a) {
  return a;
}
```

points-to

foo:a	new A1()
bar:a	new A2()

objects represented by allocation sites

what objects can a variable point to?

program

```
void foo() {
  a = new A1();
  b = id(a);
}

void bar() {
  a = new A2();
  b = id(a);
}

A id(A a) {
  return a;
}
```

points-to

foo:a	new A1()
bar:a	new A2()
id:a	new A1(), new A2()

objects represented by allocation sites

what objects can a variable point to?

program

```
void foo() {
  a = new A1();
  b = id(a);
}

void bar() {
  a = new A2();
  b = id(a);
}

A id(A a) {
  return a;
}
```

points-to

objects represented by allocation sites

foo:a	new A1()
bar:a	new A2()
id:a	new A1(), new A2()
foo:b	new A1(), new A2()
bar:b	new A1(), new A2()

points-to analysis

what objects can a variable point to?

program

```
void foo() {
  a = new A1();
  b = id(a);
}

void bar() {
  a = new A2();
  b = id(a);
}

A id(A a) {
  return a;
}
```

points-to

objects represented by allocation sites

foo:a	new A1()
bar:a	new A2()
id:a	new A1(), new A2()
foo:b	new A1(), new A2()
bar:b	new A1(), new A2()

context-sensitive points-to

foo:a	new A1()
bar:a	new A2()
id:a (foo)	new A1()
id:a (bar)	new A2()
foo:b	new A1()
bar:b	new A2()

why exception analysis? (3)

points-to analysis

- necessity: sound points-to analyses need to handle all language constructs
- exception analysis is different, and complicates points-to algorithms

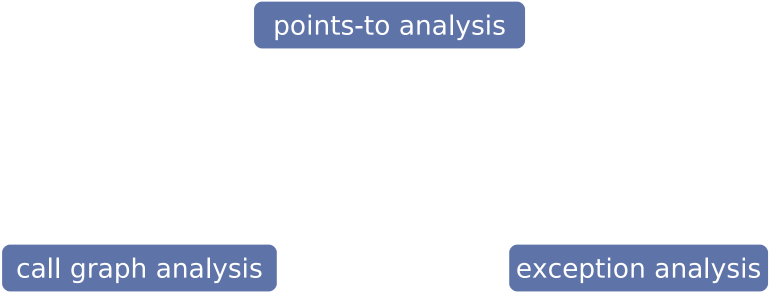
workaround: imprecise exception analysis

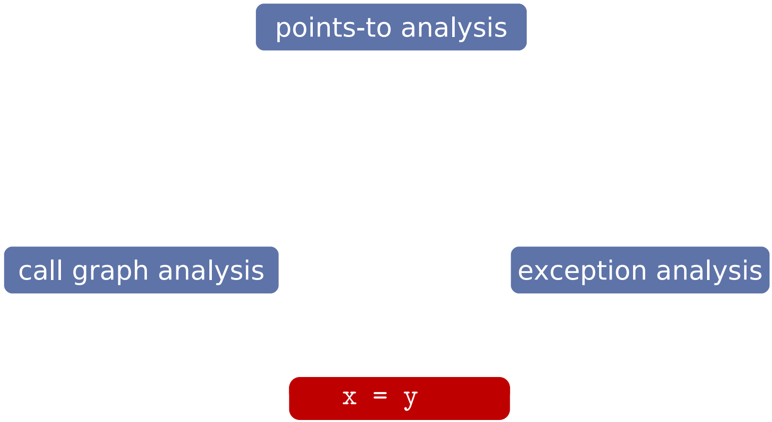
```
throw e;               ⇒  THROWN_EXCEPTIONS = e;
catch(Exception e);    ⇒  e = THROWN_EXCEPTIONS;
```

sneak preview: our finding

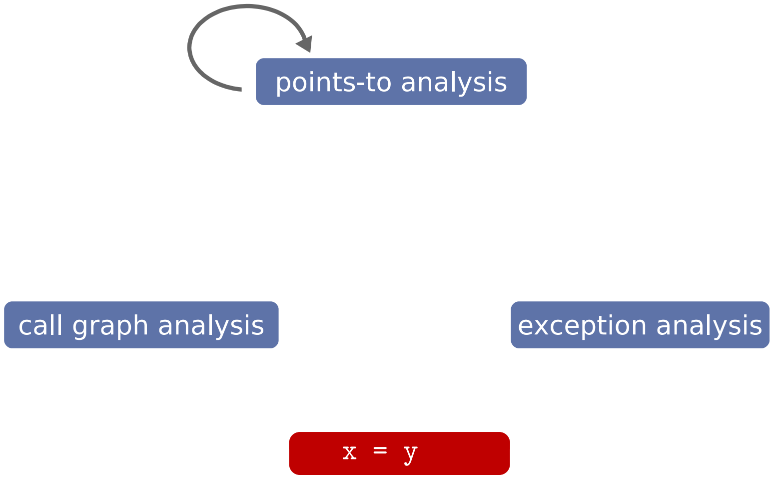
imprecise exception handling dominates the output of precise context-sensitive points-to analysis

program analysis: a domain of mutual recursion

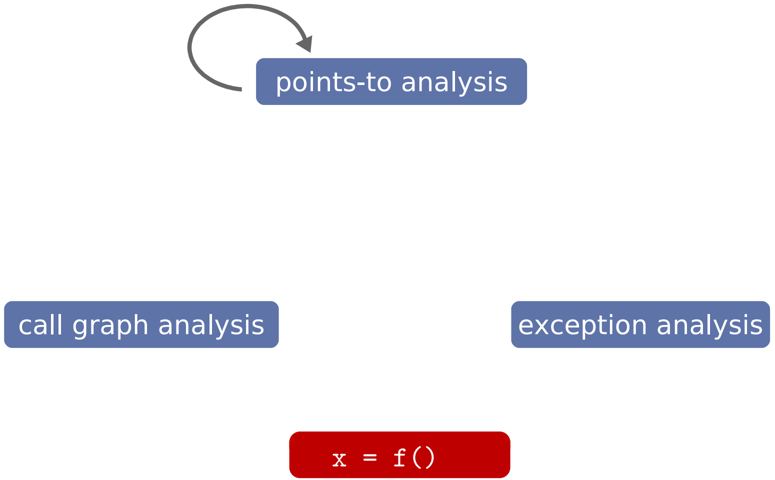


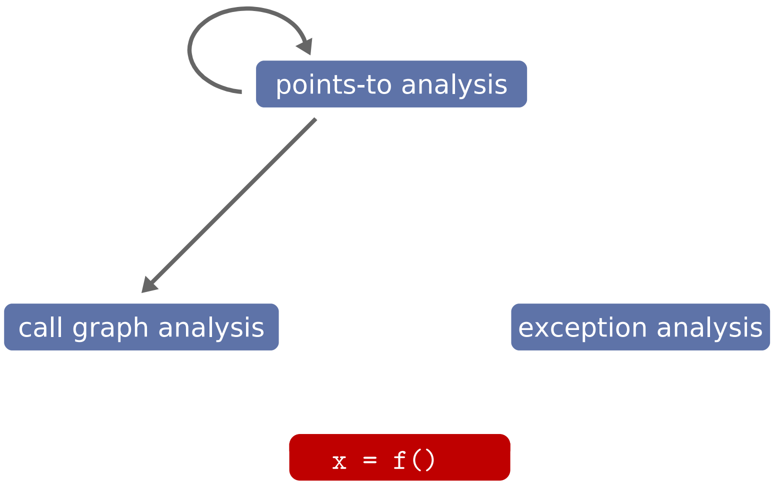

points-to analysis
call graph analysis
exception analysis
x = y

program analysis: a domain of mutual recursion

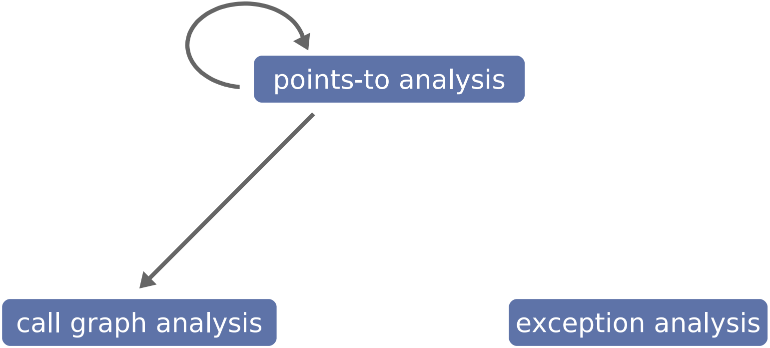


program analysis: a domain of mutual recursion

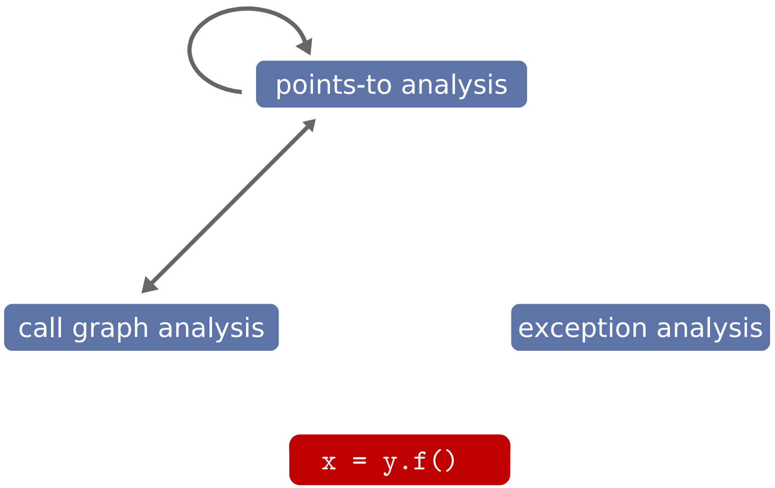



points-to analysis
call graph analysis
exception analysis
x = f()

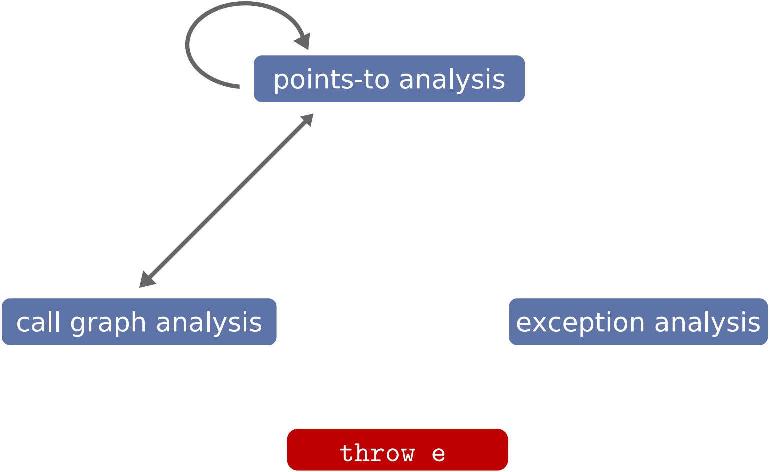
program analysis: a domain of mutual recursion

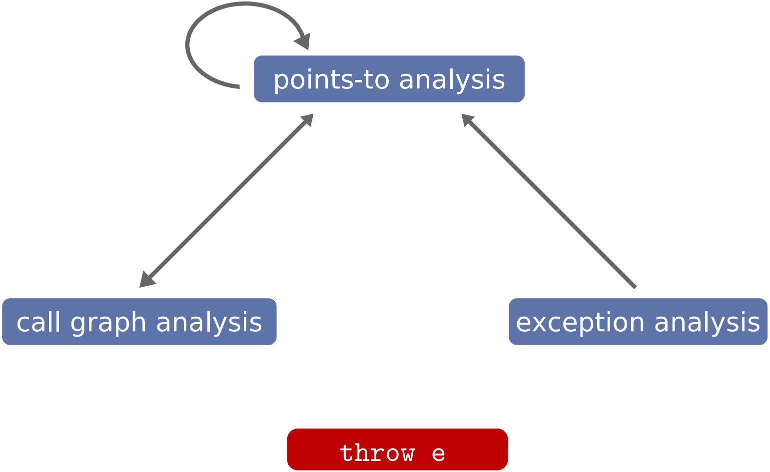


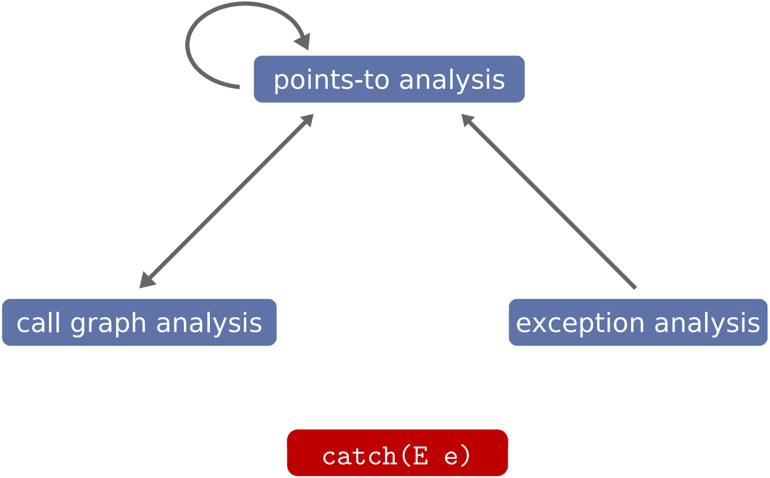

```
x = y.f()
```

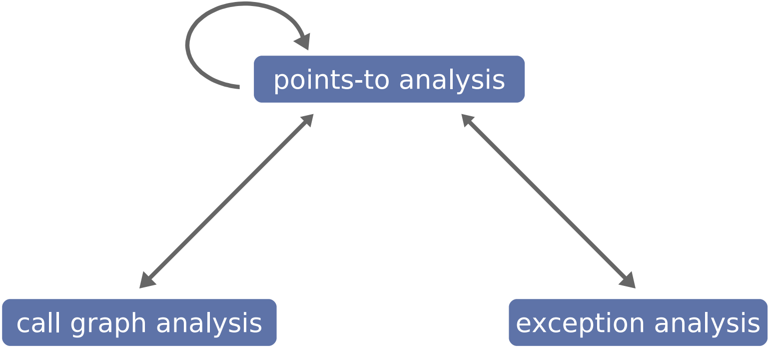

points-to analysis
call graph analysis
exception analysis
x = y.f()

program analysis: a domain of mutual recursion

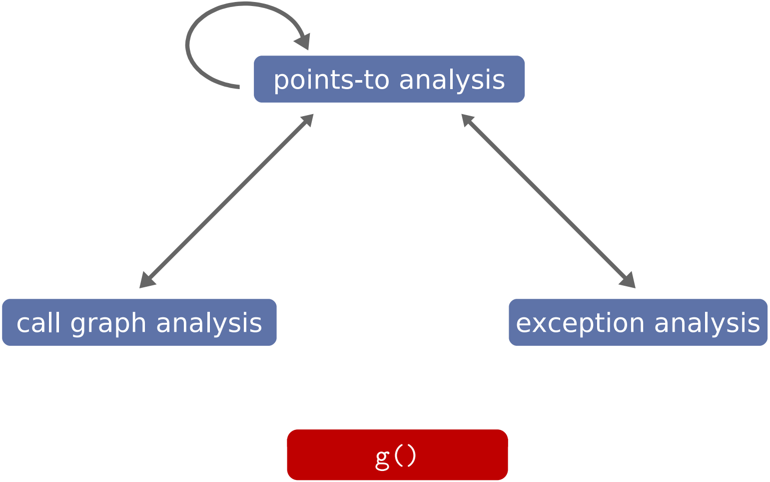



points-to analysis
call graph analysis
exception analysis
throw e

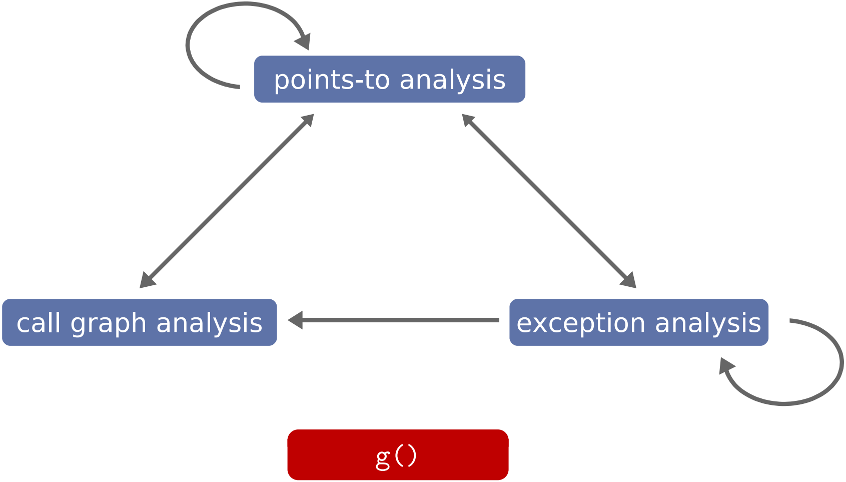

points-to analysis
call graph analysis
exception analysis
catch(E e)

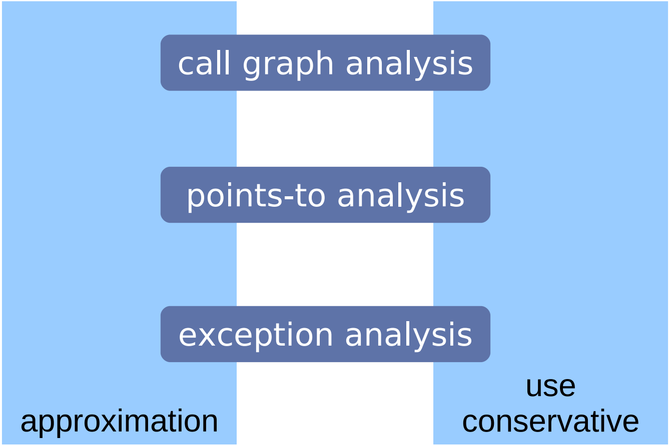

points-to analysis
call graph analysis
exception analysis

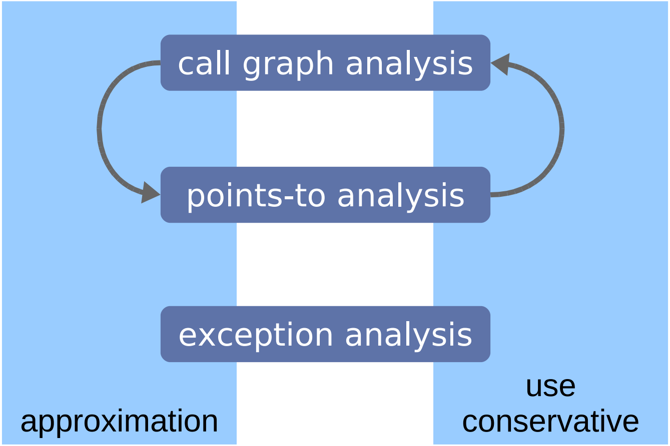
`catch(E e)`

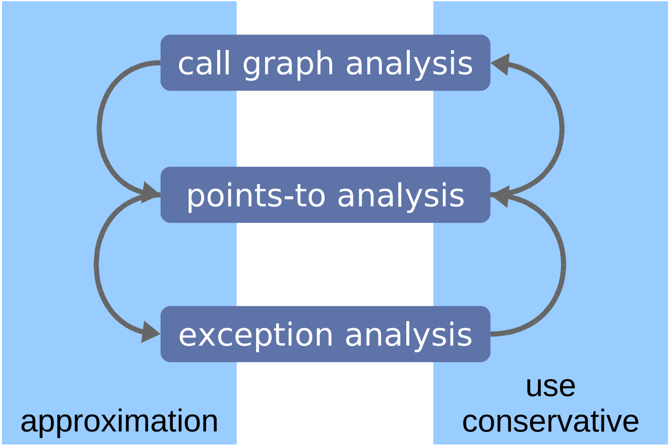

points-to analysis
call graph analysis
exception analysis
g()

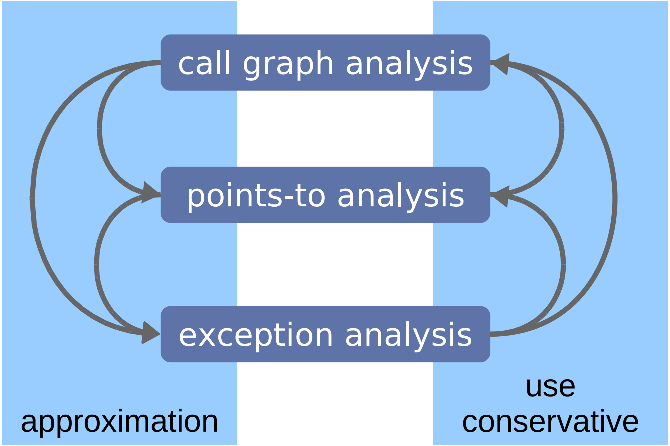
program analysis: a domain of mutual recursion

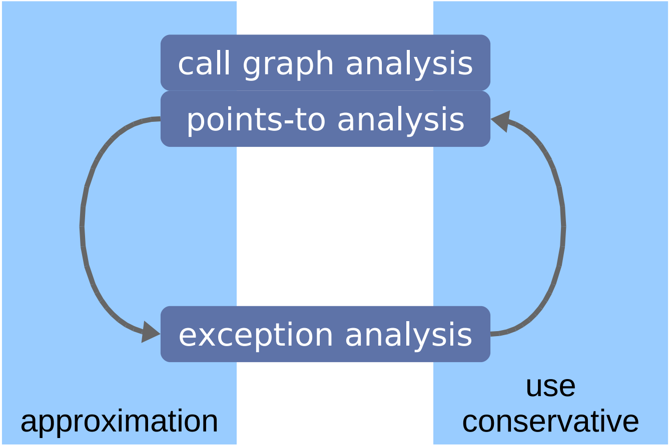


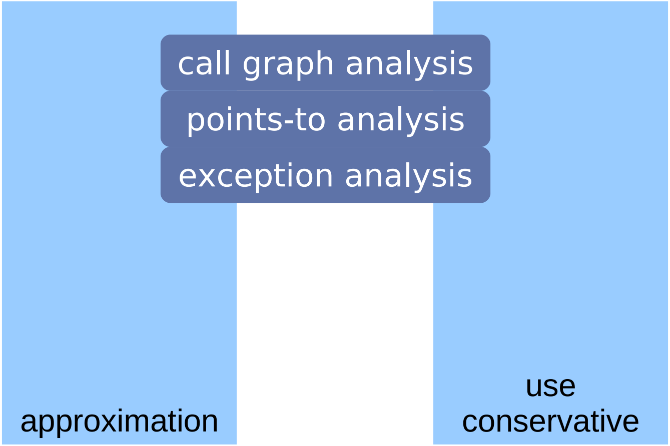

call graph analysis
points-to analysis
exception analysis
approximation
use
conservative


call graph analysis
points-to analysis
exception analysis
approximation
use
conservative


call graph analysis
points-to analysis
exception analysis
approximation
use
conservative


call graph analysis
points-to analysis
exception analysis
approximation
use
conservative


call graph analysis
points-to analysis
exception analysis
approximation
use
conservative


call graph analysis
points-to analysis
exception analysis
approximation
use
conservative

joint exception analysis and points-to analysis

- major improvement in overall precision
- major performance improvement

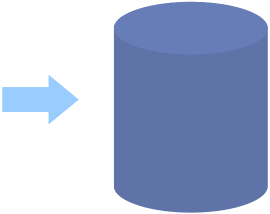
where is the magic?

- our approach: no imperative algorithm, only declarative specification
- simple declarative specification of highly complex mutually recursive dependencies in datalog

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```



datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

a	new A()
b	new B()
c	new C()

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

a	new A()
b	new B()
c	new C()

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

datalog: declarative mutual recursion

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

a	new A()
b	new B()
c	new C()
a	new B()

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

source

```
a = new A();
b = new B();
c = new C();
a = b;
b = a;
c = b;
```

AssignObjectAllocation

a	new A()
b	new B()
c	new C()

Assign

b	a
a	b
b	c

VarPointsTo

a	new A()
b	new B()
c	new C()
a	new B()
b	new A()
c	new B()
c	new A()

```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).
```

limited logic programming

- sql with recursion
  prolog without complex terms (constructors)
- captures PTIME complexity class

strictly declarative

- as opposed to prolog
  - conjunction commutative
  - rules commutative
- increases optimization opportunities
  - enables different execution strategies
  - enables more aggressive optimization

writing datalog is less programming, more specification

Strictly Declarative Specification of Sophisticated Points-to Analyses

- performance
- scalability
- declarative specification
- no BDDs

http://doop.program-analysis.org

method invocations: propagated exceptions

```
void f() {
  ...
}
```

method invocations: propagated exceptions

```
void f() {
  g();
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
```

Method declaration ?caller may throw exception object ?obj

```
void f() {
  g();
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
```

```
void f() {
  g();
}
```

Method invocation ?invocation may invoke method ?tomethod

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
```

```
void f() {
  g();
}
```

Method declaration ?tomethod may throw exception object ?obj

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
```

```
void f() {
  g();
}
```

The type of the object allocated at ?obj is ?objtype

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
```

```
void f() {
  g();
}
```

Exceptions of specific type ?objtype, thrown at instruction ?invocation, are handled by exception handler ?handler

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Instruction:Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

Instruction ?invocation is in method ?caller

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Instruction:Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
void f() {
  try {...}
  catch(E e) {...}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Instruction:Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
void f() {
  try { g(); }
  catch(E e) {...}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Instruction:Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
```

```
void f() {
  try { g(); }
  catch(E e) {...}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Instruction:Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
```

```
void f() {
  try { g(); }
  catch(E e) {...}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Instruction:Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  ExceptionHandler[?objtype, ?invocation] = ?handler,
```

```
void f() {
  try { g(); }
  catch(E e) {...}
}
```

method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Object:Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Instruction:Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  ExceptionHandler[?objtype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

```
void f() {
  try { g(); }
  catch(E e) {...}
}
```

what did you just see here?

- modular extension of variety of base points-to analyses
- approximation only comes from points-to abstraction – exception logic as precise as possible!
- complex mutually recursive dependencies
- specified elegantly in a few lines of logic

you might wonder … does that work?!

experimental findings

major experimental findings: precision and speed

statistics highlights for **object sensitive** analysis:

- precision of points-to results

context-insensitive:	imprecise $>$ precise $\times$ 1.9
context-sensitive:	imprecise $>$ precise $\times$ 3

- size of call graph

context-insensitive:	no significant difference
context-sensitive:	1.9$\times$ to 6.1$\times$ more edges

- performance

imprecise 14$\times$, 12$\times$, 5-10$\times$, 1.8$\times$ slower

major experimental findings: precision and speed

statistics highlights for **object sensitive** analysis:

- precision of points-to results

  context-insensitive: imprecise $>$ precise $\times$ 1.9
  context-sensitive: imprecise $>$ precise $\times$ 3

- size of call graph

  context-insensitive: no significant difference
  context-sensitive: 1.9$\times$ to 6.1$\times$ more edges

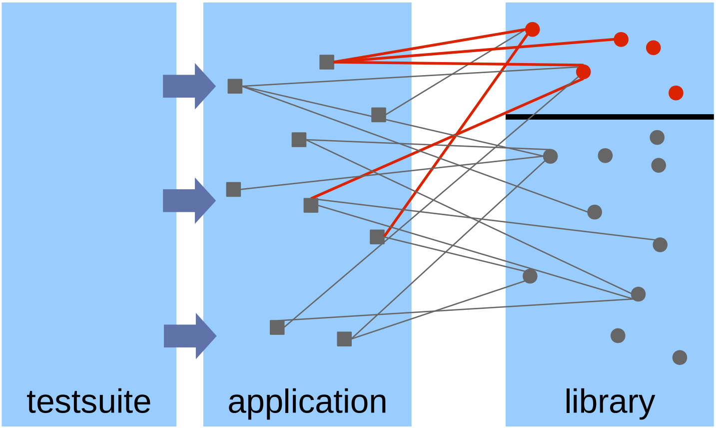
- performance

  imprecise 14$\times$, 12$\times$, 5-10$\times$, 1.8$\times$ slower

our finding

Precise exception handling has a major impact on the precision and performance of context-sensitive points-to analyses. With imprecise exception handling, the size of the problem is largely determined by exceptions.

application: test coverage of exceptional situations [Fu et al.]



test coverage: possible exception-catch links [Fu et al.]

			I/O_{sel}	time
imprecise	ftpd	insens	104	12s
		1 obj	91	23s
	muffin	insens	490	22s
		1 obj	420	86s
precise	ftpd	insens		
		1 obj		
	muffin	insens		
		1 obj		

test coverage: possible exception-catch links [Fu et al.]

			I/O_{sel}	time
imprecise	ftpd	insens	104	12s
		1 obj	91	23s
	muffin	insens	490	22s
		1 obj	420	86s
precise	ftpd	insens	47	15s
		1 obj	15	15s
	muffin	insens	237	31s
		1 obj	49	94s

test coverage: possible exception-catch links [Fu et al.]

			I/O_{sel}	time	
imprecise	ftpd	insens	104	12s	
		1 obj	91	23s	
	muffin	insens	490	22s	
		1 obj	420	86s	
precise	ftpd	insens	47	15s	custom: $\sim 5min$
		1 obj	15	15s	
	muffin	insens	237	31s	custom: $> 1h$
		1 obj	49	94s	

test coverage: possible exception-catch links [Fu et al.]

			I/O_{sel}	time	
imprecise	ftpd	insens	104	12s	
		1 obj	91	23s	
	muffin	insens	490	22s	
		1 obj	420	86s	
precise	ftpd	insens	47	15s	custom: $\sim 5min$
		1 obj	15	15s	
	muffin	insens	237	31s	custom: $> 1h$
		1 obj	49	94s	

our finding

Our **general** *joint points-to and exception analysis achieves precision comparable to a* **custom** *exception-flow analysis, but runs much faster.*

major experimental findings: approximations

selectively remove features from fully precise analysis

- order of exception handlers not considered (o)

```
catch(FileNotFoundException e) {...}
catch(IOException e) {...}
```

- no filtering of caught exceptions (f)

```
void foo() {
   try {...}
   catch(IOException e) {...}
}
```

- context-insensitive throw points-to (cs)
  - methods throw same exceptions in all contexts

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K
×		×	×1.5	×1.0	×1.1

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K
×		×	×1.5	×1.0	×1.1
×	×		×2.6	×1.2	×1.9

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K
×		×	×1.5	×1.0	×1.1
×	×		×2.6	×1.2	×1.9
×			×2.6	×1.3	×1.9

major experimental findings: approximations

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K
×		×	×1.5	×1.0	×1.1
×	×		×2.6	×1.2	×1.9
×			×2.6	×1.3	×1.9
	×	×	×1.1	×1.1	×1.9

major experimental findings: approximations

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K
×		×	×1.5	×1.0	×1.1
×	×		×2.6	×1.2	×1.9
×			×2.6	×1.3	×1.9
	×	×	×1.1	×1.1	×1.9
		×	×1.6	×1.2	×2.1
	×		×2.7	×1.4	×3.4
			×2.7	×1.5	×3.4

major experimental findings: approximations

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K
×		×	×1.5	×1.0	×1.1
×	×		×2.6	×1.2	×1.9
×			×2.6	×1.3	×1.9
	×	×	×1.1	×1.1	×1.9
		×	×1.6	×1.2	×2.1
	×		×2.7	×1.4	×3.4
			×2.7	×1.5	×3.4
imprecise			×6.1	×2.0	-

major experimental findings: approximations

cs	o	f	call graph edges	var points-to	throw points-to
×	×	×	1.0M	598K	579K
×		×	×1.5	×1.0	×1.1
×	×		×2.6	×1.2	×1.9
×			×2.6	×1.3	×1.9
	×	×	×1.1	×1.1	×1.9
		×	×1.6	×1.2	×2.1
	×		×2.7	×1.4	×3.4
			×2.7	×1.5	×3.4
imprecise			×6.1	×2.0	-

our finding

Every approximation of exception handling significantly increases var points-to, throw points-to, or call graph edges.

major experimental findings

points-to analysis

Precise exception handling has a major impact on the precision and performance of context-sensitive points-to analyses.

exception-flow analysis

Our *general* joint points-to and exception analysis achieves precision comparable to a *custom* exception-flow analysis, but runs much faster.

approximations

Every approximation of exception handling significantly increases var points-to, throw points-to, or call graph edges.

type-based exception analyses [Robillard, Jex]

- do not determine where an exception comes from
- conservative/unsound for ‘computed’ exceptions

type-based exception analyses [Robillard, Jex]

- do not determine where an exception comes from
- conservative/unsound for ‘computed’ exceptions

exception-flow and exception-chain analysis [Fu et al.]

- precise analysis
- slow, automatically supported by points-to analysis

related work

type-based exception analyses [Robillard, Jex]

- do not determine where an exception comes from
- conservative/unsound for ‘computed’ exceptions

exception-flow and exception-chain analysis [Fu et al.]

- precise analysis
- slow, automatically supported by points-to analysis

spark, paddle [Lhotak et al.], **bddbddb** [Whaley et al]

- imprecise exception analysis
- generally not integrated in the analysis

related work

type-based exception analyses [Robillard, Jex]

- do not determine where an exception comes from
- conservative/unsound for ‘computed’ exceptions

exception-flow and exception-chain analysis [Fu et al.]

- precise analysis
- slow, automatically supported by points-to analysis

spark, paddle [Lhotak et al.], **bddbddb** [Whaley et al]

- imprecise exception analysis
- generally not integrated in the analysis

doop compared to other datalog-based points-to analysis

- full end-to-end analysis in datalog
- first precise declarative exception analysis

what have we seen?

- joint points-to and exception analysis

what have we seen?

- joint points-to and exception analysis
- precision of exception analysis has significant impact on points-to analysis

conclusion

what have we seen?

- joint points-to and exception analysis
- precision of exception analysis has significant impact on points-to analysis
- exception analysis as precise, but much faster than custom exception analyses

conclusion

what have we seen?

- joint points-to and exception analysis
- precision of exception analysis has significant impact on points-to analysis
- exception analysis as precise, but much faster than custom exception analyses

what more is in the paper?

- computing exception handlers
- experiments
- background on datalog and points-to analysis