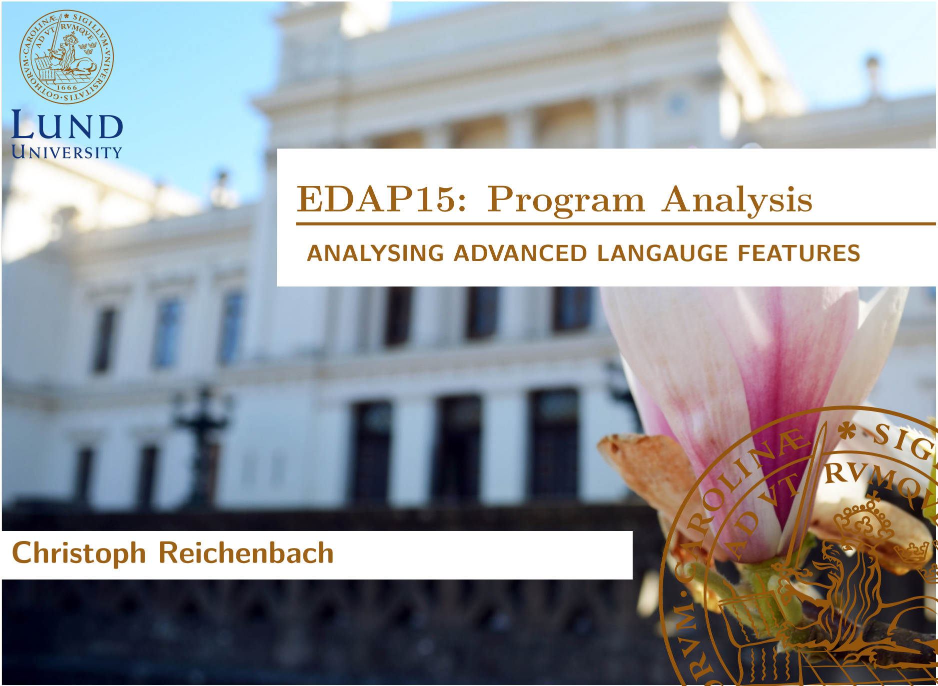


LUND UNIVERSITY

EDAP15: Program Analysis

ANALYSING ADVANCED LANGAUGE FEATURES

Christoph Reichenbach

Welcome back!

- Questions?

Applying IFDS to Java

Java

```
public static void main(String[] args) {
    Object obj = MyClass.getObj();
    System.err.println(obj.toString());
}
```

Subroutine call

- Analogous to Teal-0 calls
- . . . need to know `MyClass`

Method call

- **Dynamic Dispatch**
- Exact subroutine depends on *dynamic type* of `obj`

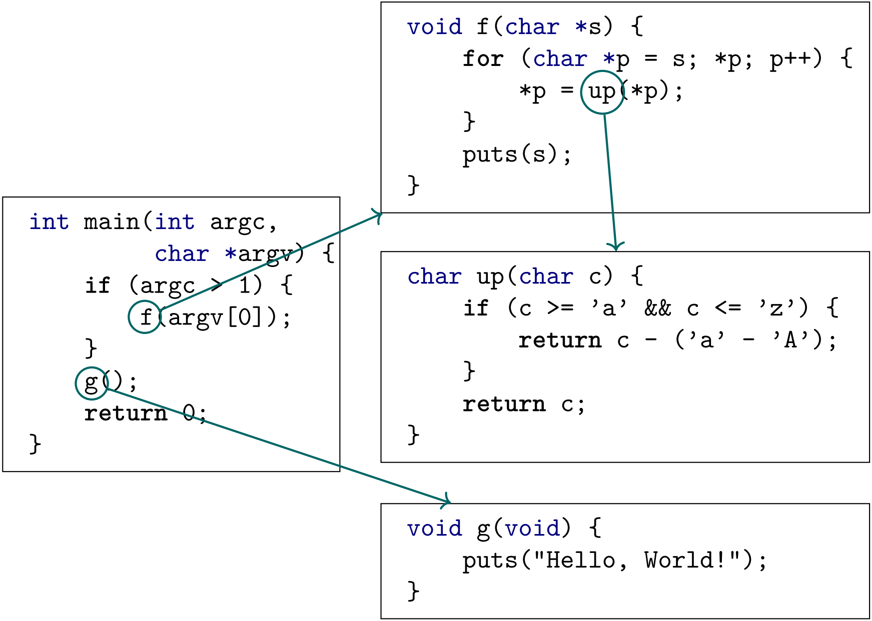
Challenges

- **Other modules**:
  - Must have access to analysable representation of module
  - *Not always available*
- **Dynamic Dispatch**:

```
obj.toString()
```

  - Which `toString` method are we calling?
  - Worst case assumption: *any* class (`Integer.toString()`, `HashSet.toString()`, ...)
  - Can we do better?

The Call Graph



```
int main(int argc,
         char *argv) {
    if (argc > 1) {
        f(argv[0]);
    }
    g();
    return 0;
}
```

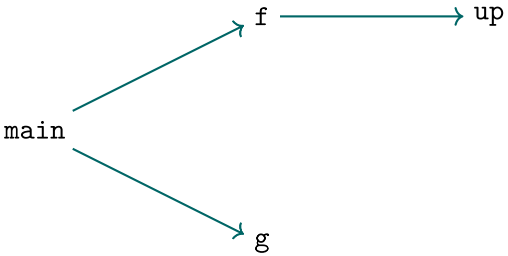
```
void f(char *s) {
    for (char *p = s; *p; p++) {
        *p = up(*p);
    }
    puts(s);
}
```

```
char up(char c) {
    if (c >= ’a’ && c <= ’z’) {
        return c - (’a’ - ’A’);
    }
    return c;
}
```

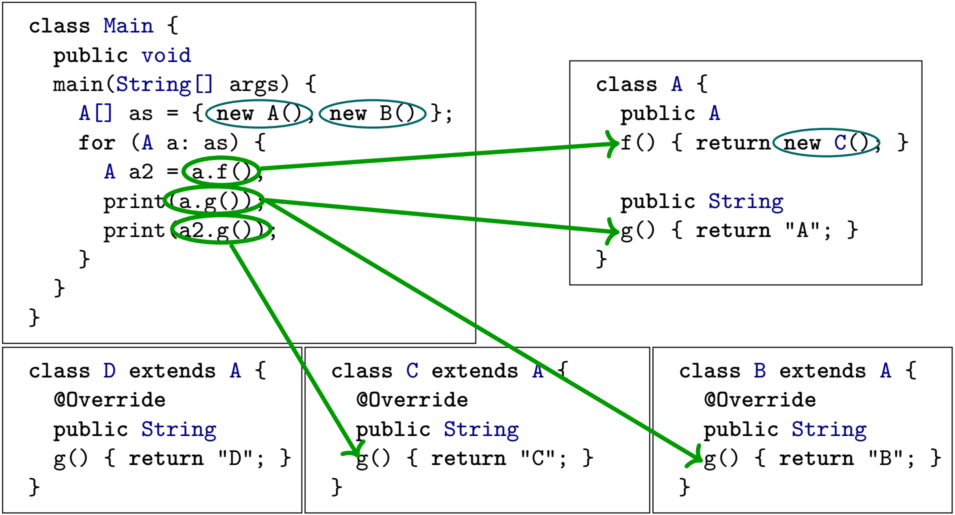
```
void g(void) {
    puts("Hello, World!");
}
```

The Call Graph

- $G_{\text{call}} = \langle P, E_{\text{call}} \rangle$
- Connects procedures from P via call edges from E_{call}
- 'Which procedure can call which other procedure?'
- Often refined to:
  'Which *call site* can call which procedure?'
- Used by program analysis to find procedure call targets



Finding Calls and Targets



```
class Main {
  public void
  main(String[] args) {
    A[] as = {new A(), new B()};
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

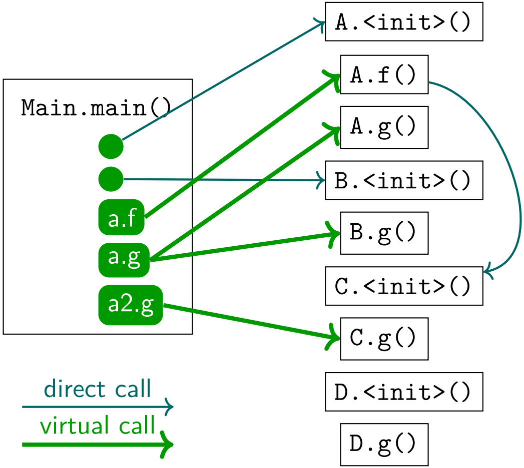
```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Dynamic Dispatch: Call Graph

Challenge: Computing the precise call graph:



Summary

- **Call Graphs** capture which procedure calls which other procedure
- For program analysis, further specialised to map:

$$\text{Callsite} \rightarrow \text{Procedure}$$

- Direct calls: straightforward
- Virtual calls (dynamic dispatch):
  - Multiple targets possible for call
  - No fully sound/precise solution in general

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

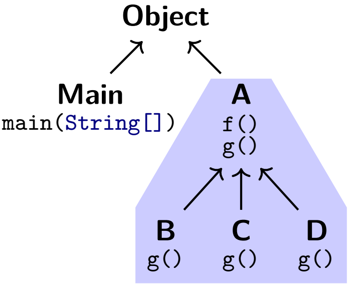
```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

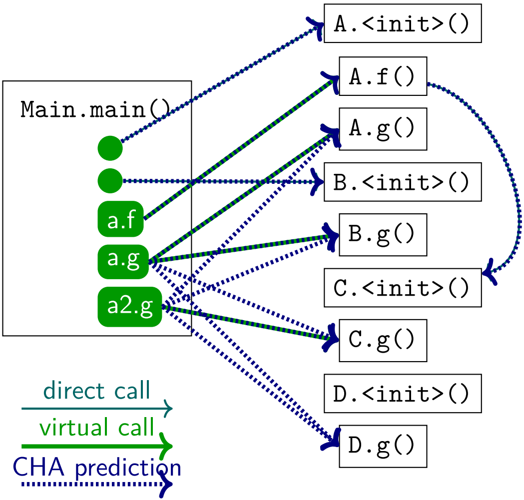
```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Class Hierarchy Analysis



- Use declared type to determine possible targets
- Must consider all possible subtypes
- In our example: assume a.f can call any of: `A.f()`, `B.f()`, `C.f()`, `D.f()`

Class Hierarchy Analysis: Example



Summary

- **Call Hierarchy Analysis** resolves virtual calls $a.f()$ by:
  - Examining static types T of receivers $(a : T)$
  - Finding all subtypes $S <: T$
  - Creating call edges to all $S.f$, if $S.f$ exists
- **Sound**
  - Assuming strongly and statically typed language with subtyping
- Not very **precise**
  - Java: `((Object) obj).toString()`:
    Will use *all* `toString()` methods *anywhere*

Rapid Type Analysis

- Intuition:
    - Only consider reachable code
    - Ignore unused classes
    - Ignore classes instantiated only by unused code

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

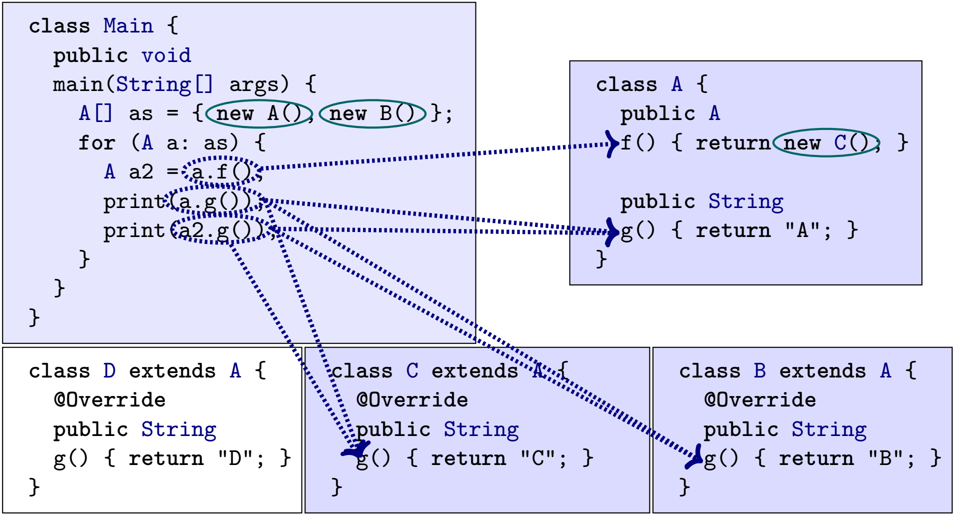
```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Finding Calls and Targets



```
class Main {
  public void
  main(String[] args) {
    A[] as = {new A(), new B()};
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

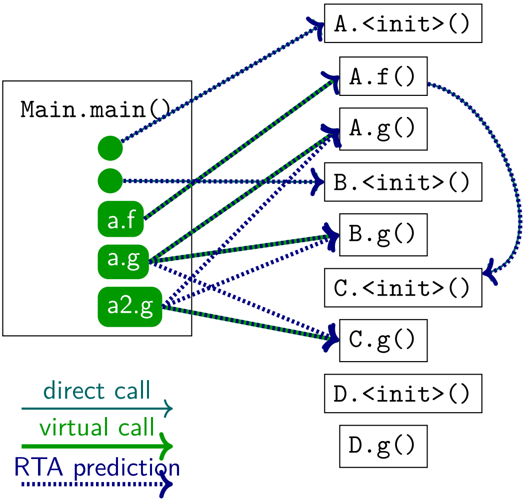
```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Rapid Type Analysis: Example



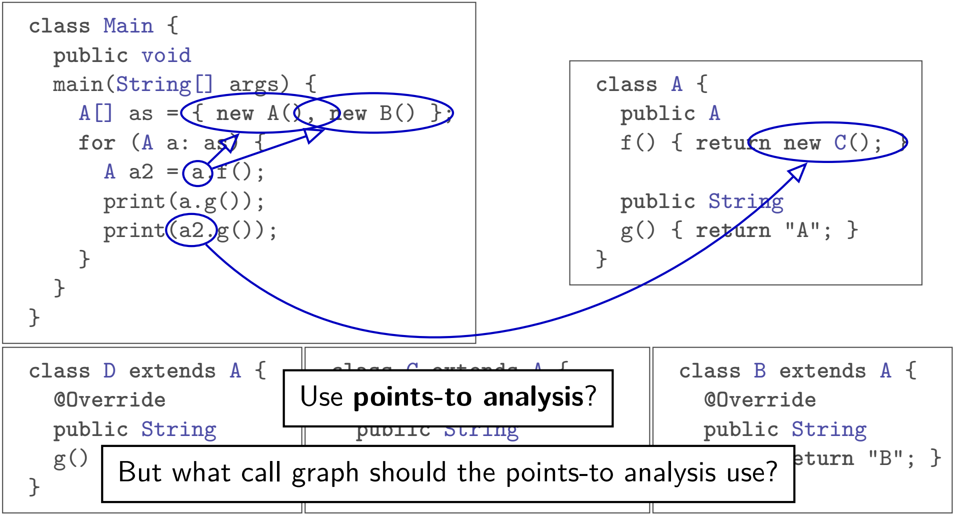
Rapid Type Analysis Algorithm Sketch

```
Procedure RTA(mainproc, <:):
begin
  WORKLIST := {mainproc}
  VIRTUALCALLS := ∅
  LIVECLASSES := ∅
  while $s \in$ mainproc do
    foreach call $c \in s$ do
      if c is direct call to $p$ then
        addToWorklist($p$)
        registerCallEdge($c \rightarrow p$)
      else if c = v.m() and $v : T$ then begin
        VIRTUALCALLS := VIRTUALCALLS ∪ {$c$}
        foreach $S <: T$ do
          addToWorklist(S.m)
          registerCallEdge($c \rightarrow$ S.m)
        done
      end else if c = new $C$() and $C \notin$ LIVECLASSES then begin
        LIVECLASSES := LIVECLASSES ∪ {$C$}
        foreach $v.m() \in$ VIRTUALCALLS with $v : T$ and $C <: T$ do
          addToWorklist(C.m)
          registerCallEdge($c \rightarrow$ C.m)
        done
      end
done done end
```

Summary

- **Rapid Type Analysis** resolves virtual calls $a.f()$ as follows:
  - Find all classes that can be instantiated in reachable code
  - Expand reachable code:
    - For direct calls to p, add p as reachable
    - For all virtual calls to $v.m()$ with $v : T$:
      $\Rightarrow$ Add $S.m()$ as reachable
  - Iterate until we reach a fixpoint
- **Sound**
  - Assuming strongly and statically typed language with subtyping
- More **precise** than Class Hierarchy Analysis

Finding Calls and Targets



```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g()
}
```

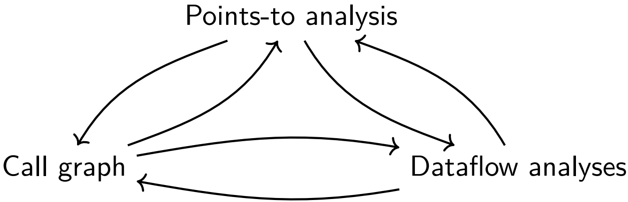
```
class C extends A {

  public String
```

```
class B extends A {
  @Override
  public String
           eturn "B"; }
```

Use **points-to analysis**?

But what call graph should the points-to analysis use?

Dependencies



- Mutual dependencies across program analyses

Loose Composition

Loose Composition: **Split analyses into multiple passes**

- Each pass finishes before next pass starts
- Example:
  1. **RTA**: compute initial call graph
  2. **Steensgaard** on RTA output: conservative points-to graph
  3. Build **pointer-based call graph** from Steensgaard's results
  4. **Andersen's analysis** with refined (smaller) call graph

Tight Composition

Tight Composition: **Analyses depend on each other's intermediate results**

- Analyses run "together"
- Example:
  - JastAdd circular attribute computations (Exercise 2)
  - Could combine data flow analysis with points-to or call-graph analysis
- **Challenges**:
  - Traditional worklist algorithms:
    - Complex manual engineering needed
  - Declarative approaches:
    - Must guarantee **Monotonicity**

Summary

- Mutual dependency between *points-to*, *data flow*, *call graph* analyses
- Two approaches:
  - **Loose composition**:
    - One analysis after the other
    - May need to run analyses multiple times
  - **Tight composition**:
    - Analyses can use each other's intermediate results
    - Difficult to engineer for worklist algorithms
    - Easier with declarative approaches (attribute grammars, logic programming)

Summary: Flow-Insensitive Analysis

- **Monomorphic type inference**
  - Free variables, occurs check, unification
  - Close to $O(\#\text{AST nodes})$
- Polymorphic type inference (Hindley-Damas-Milner)
  - Type schemas and instantiation
  - DEXPTIME-complete
- **Steensgaard's points-to analysis**
  - Similar to monomorphic type inference
  - Close to $O(\#\text{AST nodes})$
- **Andersen's points-to analysis**
  - Points-to edges and inclusion edges that generate new edges
  - $O(\#\text{nodes}^3)$

Summary: Data Flow Analyses

- **MFP**
  - Precise for distributive frameworks
  - $O(\#edges \times height(\mathcal{L}))$
- **MOP**
  - Precise for monotone frameworks
  - Undecidable
- **IFDS** / IDE
  - Interprocedural, precise for distributive frameworks
  - $O(\#\text{edges} \times \#\text{variables}^3)$
    (IDE: $O(\#\text{edges} \times \#\text{variables}^3 \times height(\mathcal{L}))$)

Summary: Call Graph Analyses

- **Class Hierarchy analysis**
  - Trivial
  - $O(\#\text{classes} \times \#\text{methods})$
- **Rapid Type Analysis**
  - Transitive reachability check
  - $O(\#\text{classes} \times \#\text{methods})$
- **Points-to-based call graph analysis**
  - Mutual dependency
  - Complexity and precision vary

Building Analyses: Considerations

- What level of soundness?
  - Conservative: sound, but can be imprecise
  - Optimistic: unsound, but can be more precise
- What performance needs?
  - Trade-off: soundness vs. precision vs. performance
  - More precise server analysis $\implies$ faster client analysis
  - Some analyses can be split into:
    - fast/coarse "filter" pass
    - slow/precise main pass
  - Interactive use? Low latency, consider incremental analyses
  - High reliability need? (Integrate interactive tools?)

  ...
- What do we know?
  - Language semantics
  - External libraries of importance
  - User annotations / specs to help analysis

  ...

Points-to-Analysis Sensitivities

- Points-to analysis is nondistributive
  - No easy route to precise interprocedural analysis
  - No known effective procedure summary representation
- We still want non-distributive analyses to be precise
  - Example: out-of-bounds checking in method-of-interest `copy()` needs size of array (assumption: we need array allocation site)
  - Approach: repeat analysis on same code for multiple *contexts*
    - no bounds violation in copy at $\mathcal{C}_0$
    - bounds violation in copy at $\mathcal{C}_1 \Leftarrow \mathcal{A}_3$
    - bounds violation in copy at $\mathcal{C}_2 \Leftarrow \mathcal{C}_2$

```
array0 = { }           // A0
array3 = { 0, 1, 2 } // A1
c0 = new Copier(array3) // A2
c1 = new Copier(array0) // A3
c0.copy(array3) // C0
c1.copy(array3) // C1
c0.copy(array0) // C2
```

```
class Copier {
  Copier(int[] s) {
    this.src = s
  }
  copy(int[] dest) {
    dest[0] = this.src[0]
} }
```

k-call-site Sensitivity

- Call-site sensitivity (Doop terminology; traditionally called *context-sensitivity*) analyses method once per call site
- Can determine that $\mathcal{C}_0$ is safe, $\mathcal{C}_1$ is unsafe
  - Analyses `get0` twice: Two different contexts $\mathcal{C}_0$ and $\mathcal{C}_1$
- Simple call-site sensitivity *cannot* distinguish $\mathcal{C}_2$ and $\mathcal{C}_3$
  - Will only analyse `get0` once, for context $\mathcal{C}_4$
- 2-call-site sensitivity: extend context to *caller's caller*
  - Contexts: $\langle \mathcal{C}_2, \mathcal{C}_4 \rangle$ and $\langle \mathcal{C}_3, \mathcal{C}_4 \rangle$
- Need 3-call-site sensitivity etc. for deeper calls

```
array0 = {}
v = get0({ 0, 1 }) // C0
v = get0(array0)   // C1
v = f({ 0, 1 })    // C2
v = f(array0)      // C3
v = g({ 0, 1 })    // C5
```

```
int get0(int[] array) {
  return array[0] }
int f(int[] array) {
  return get0(array) // C4 }
int g(int[] array) {
  return f(array) // C6 }
```

Summary

- Analysis sensitivities allow us to analyse methods more precisely
  - Multiple analyses of same method in different *contexts*
  - Context provides additional information (args, globals, heap)
  - With procedure summaries (cf. IFDS / IDE): no repeat analysis necessary, but *only for distributive frameworks*
- **Call site sensitivity** (traditionally called *context sensitivity*) uses call sites as context
- k-**call site sensitivity** for $k > 1$ uses call sites, parent call sites, grandparent call sites etc. as context
- Other approaches:
  - *Object sensitivity* uses abstract receiver objects
  - *Plain k-object sensitivity* for $k > 1$ abstract receiver objects of the ancestor method call(s)
  - *Full k-object sensitivity* uses abstract receiver objects of the ancestor method call(s) for current object's constructor call
  - *Type sensitivity* abstracts over full k-object sensitivity by merging call sites from same type
  - Worst case analysis cost exponential over k

Analysing Realistic Programs

- Multiple analyses
- Mutual dependency between analyses
- Challenges:
  - IFDS (fast, scalable) needs distributive framework
  - Pointer analysis is:
    - crucial
    - Either imprecise or slow
    - not distributive
  - Making non-distributive analyses precise may require:
    - call-site sensitivity without procedure summaries
    - Multiple levels of call-site sensitivity
    - Alternatives: object sensitivity, type sensitivity
    - Picking the right one depends on input program, libraries, frameworks
  - Language semantics may be imprecisely defined
  - Certain **language features** intrinsically hard to analyse

Reflection

Java

```
Class<?> cl = Class.forName(string);
Object obj = cl.getConstructor().newInstance();
System.out.println(obj.toString());
```

- Instantiates object by string name
- Similar features to call method by name
- **Challenge**:
  - `obj` may have *any* type ⇒ imprecision
  - Sound call graph construction very conservative
- **Approaches**
  - Dataflow: what strings flow into `string`?
    - Common: code draws from finite set or uses string prefix/suffix (e.g., ("com.x.plugins." + . . .))
    - `Class.forName`: class only from some point in package hierarchy
  - Dynamic analysis

Dynamic Loading

C

```
handle = dlopen("module.so", RTLD_LAZY);
op = (int (*)(int)) dlsym(handle, "my_fn");
```

- Dynamic library and class loading:
  - Add new code to program that was not visible at analysis time
- **Challenge**:
  - Can't analyse what we can't see
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *all that it can reach*
  - With dynamic support and static annotation:
  - Allow only loading of signed/trusted code
    - signature must guarantee properties we care about
    - annotation provides properties to static analysis
  - *Proof-carrying code*
    - Code comes with proof that we can check at run-time

Native Code

Java

```
class A {
  public native Object op(Object arg);
}
```

- High-level language invokes code written in low-level language
  - Usually C or C++
  - May use nontrivial interface to talk to high-level language
- **Challenge**:
  - High-level language analyses don't understand low-level language
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *anything*
  - Manually model known native operations (e.g., Doop)
  - Multi-language analysis (e.g., Graal)

‘eval’ and dynamic code generation

Python

```
eval(raw_input())
```

- Execute a string as if it were part of the program
- **Challenge**:
  - Cannot predict contents of string in general
- **Approaches**:
  - Conservative approximation
    - Tricky: code may modify *anything*
  - Dynamically re-run static analysis
  - Special-case handling (cf. reflection)

Summary

- Static program analysis faces significant challenges:
  - **Decidability** requires lack of precision or soundness for most of the interesting analyses
  - **Reflection** allows calling methods / creating objects given by arbitrary string
  - **Dynamic module loading** allows running code that the analysis couldn't inspect ahead of time
  - **Native code** allows running code written in a different language
  - **Dynamic code generation** and `eval` allow building arbitrary programs and executing them
  - No universal solution
  - Can try to 'outlaw' or restrict problematic features, depending on goal of analysis
  - Can combine with dynamic analyses

Soundiness

- Can't analyse language feature?

$\Rightarrow$ We get $\top$

$\Rightarrow$ Many false positives

$\Rightarrow$ Tool may be useless

  - Google SWE practice: Bug checkers with $> 5\%$ false positives disabled automatically
- Soundness may not be *useful*
- Alternative proposal: **Soundiness**
  - *Be explicit* about unsupported language features

Soundiness: "capture all dynamic behaviour *within reason*"