EDA045F: Program Analysis

LECTURE 5 BONUS: BASIC CALLGRAPHS

Christoph Reichenbach

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

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

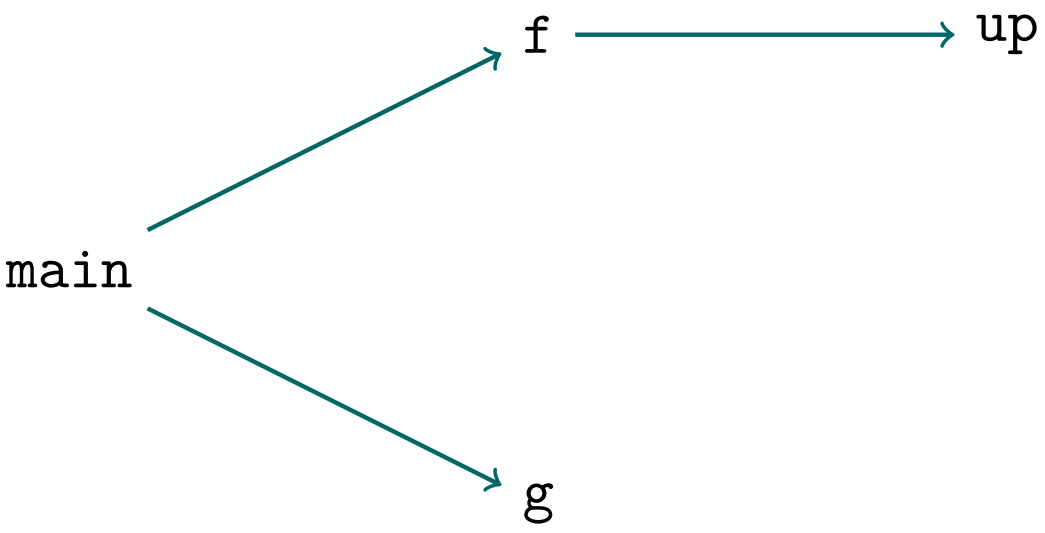
```
void f(char *s) {
    for (char *p = s; *p; p++) {
        *p = up(*p);
    }
    puts(s);
}
```

```
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 "A"; }
}
```

```
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 "A"; }
}
```

```
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"; }
}
```

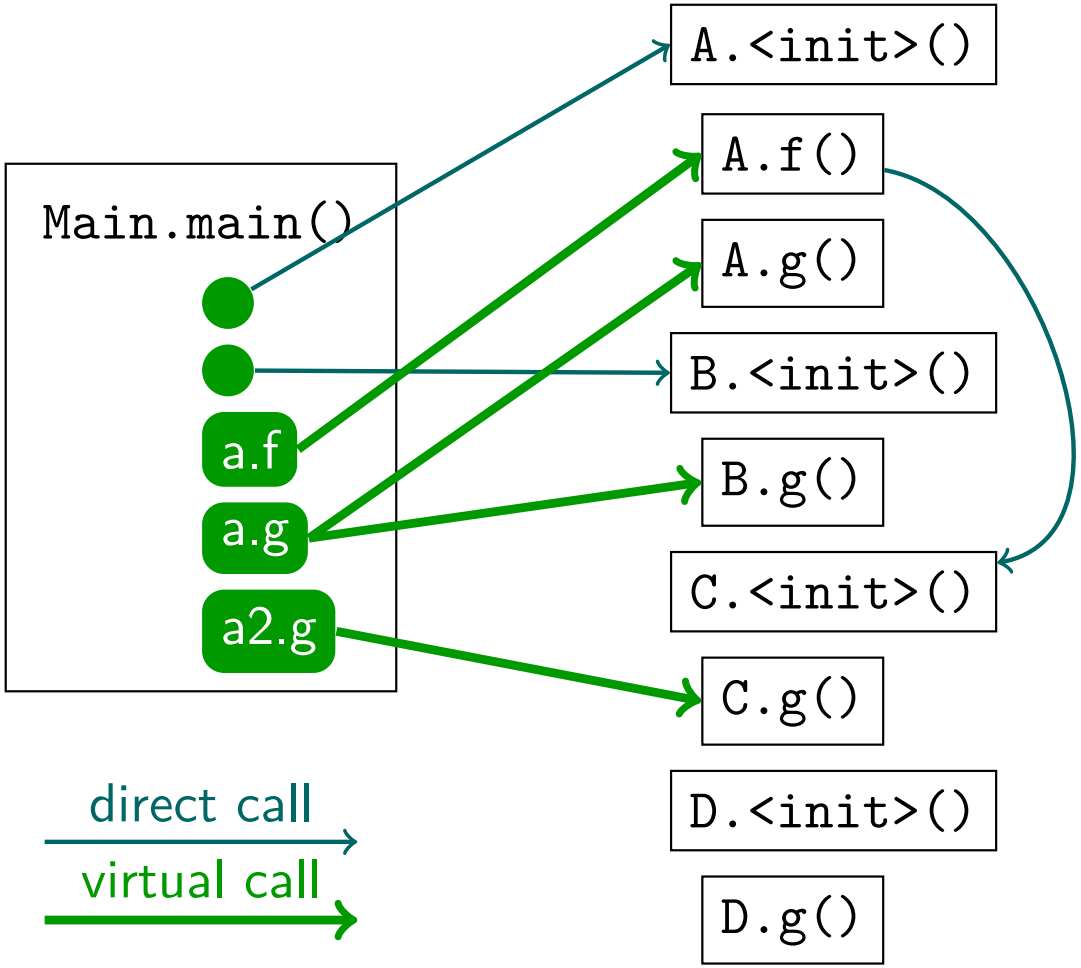
```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

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

```
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
  - Not straightforward

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 "A"; }
}
```

```
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());
    }
  }
}
```

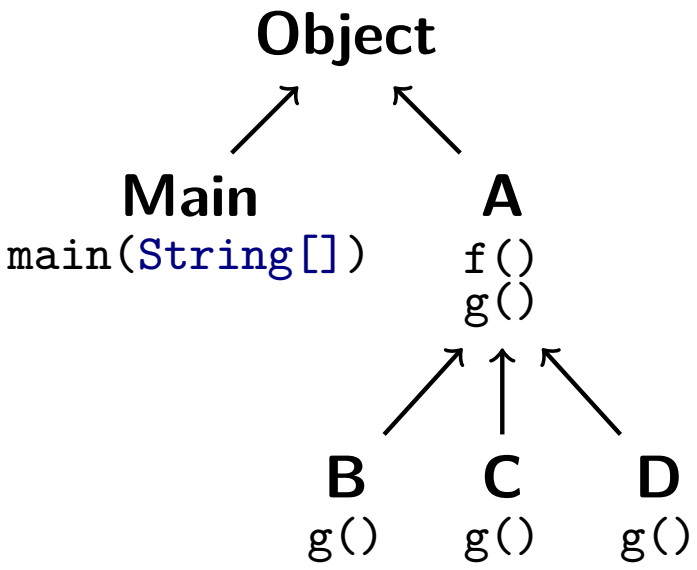
```
class A {
  public A
  f() { return new C(); }

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

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

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

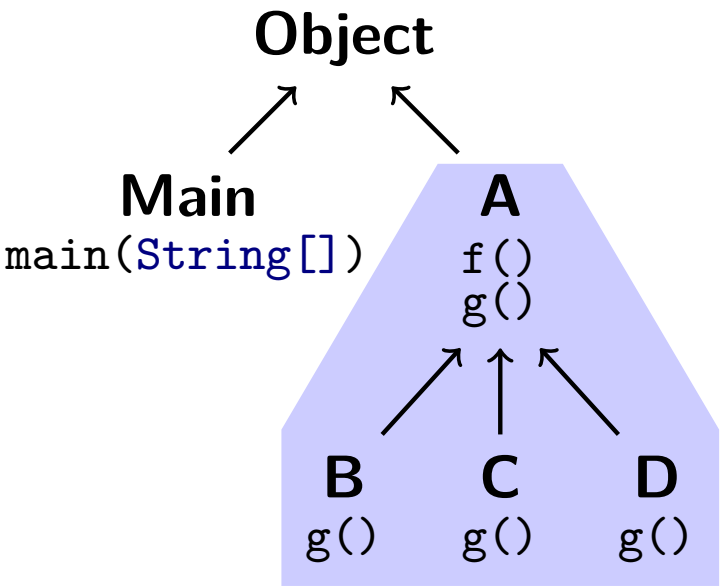
```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Class Hierarchy Analysis



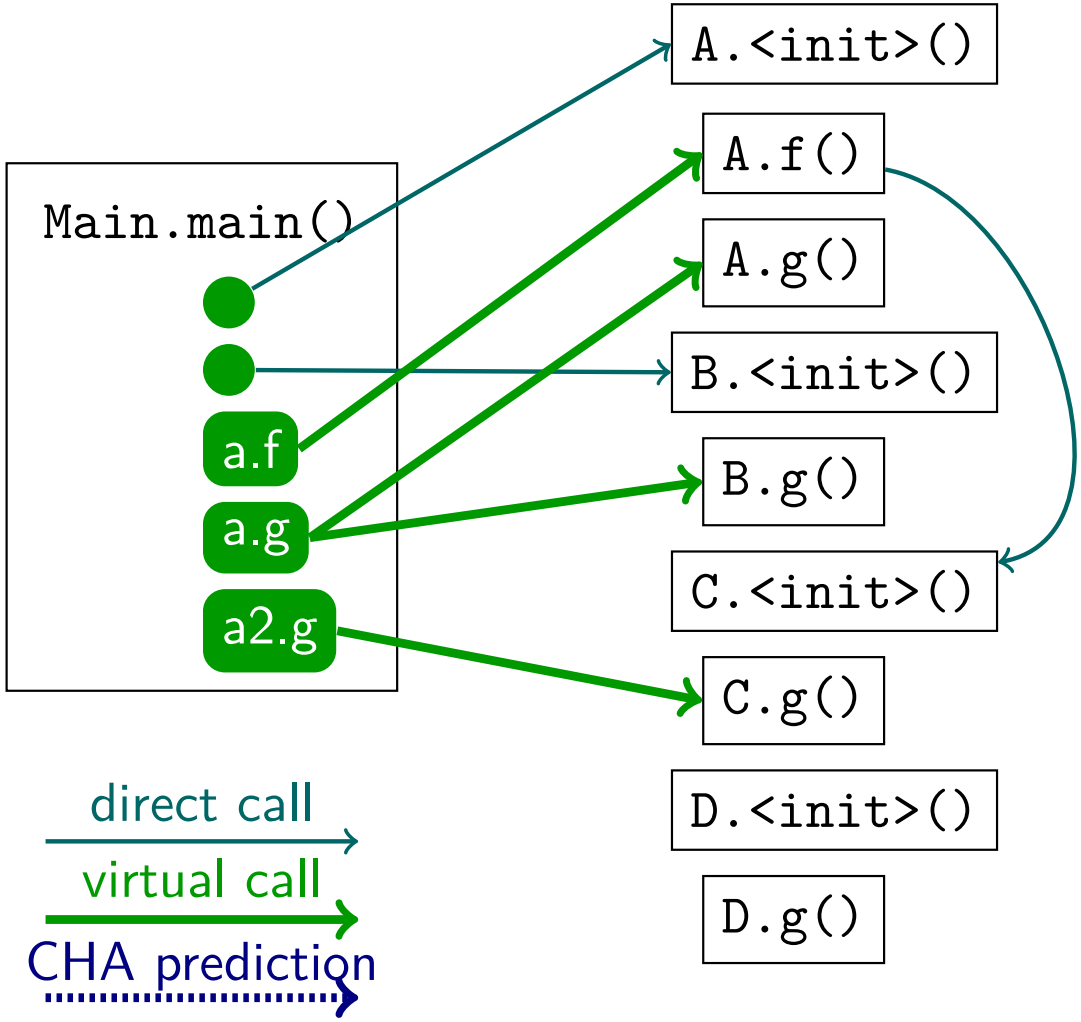
- Use declared type to determine possible targets

Class Hierarchy Analysis

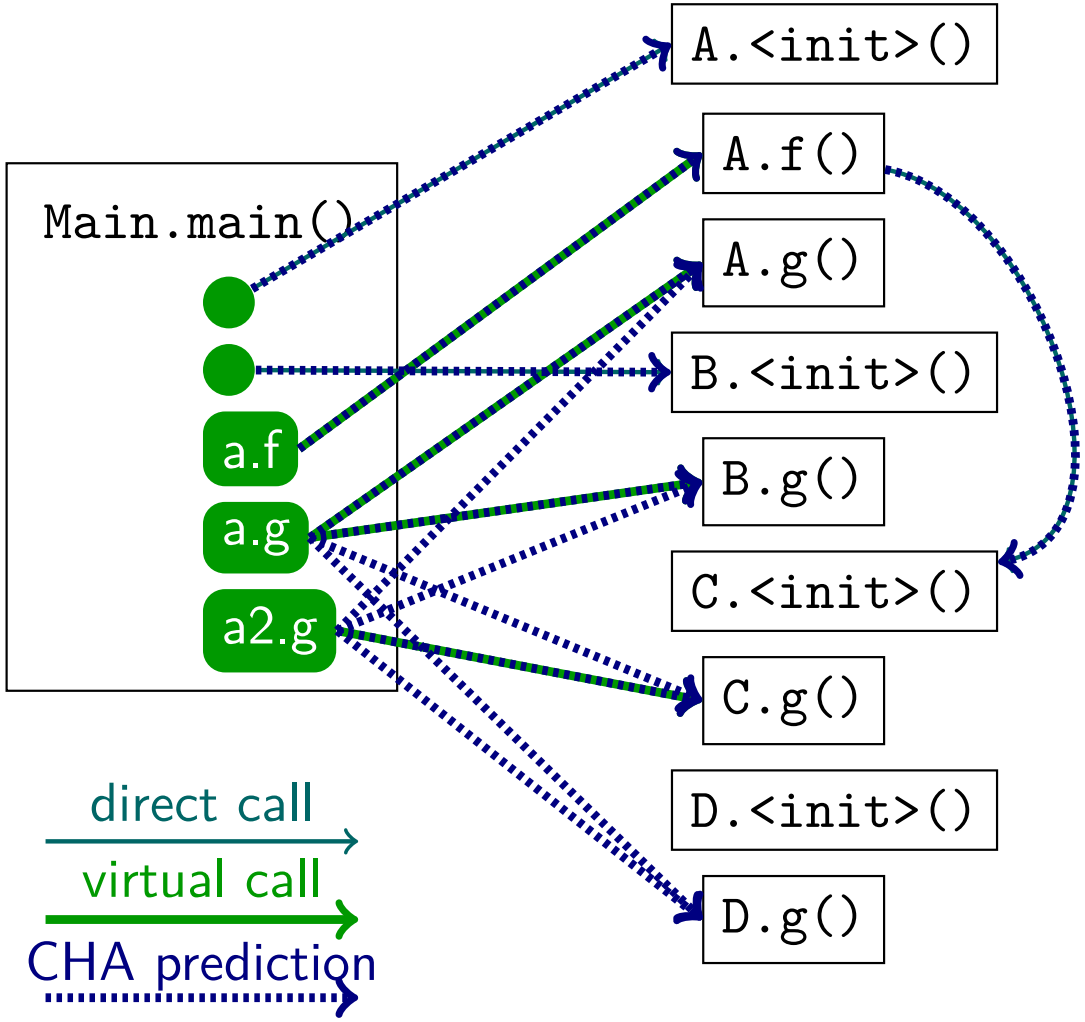


- Use declared type to determine possible targets
- Must consider all possible subtypes
- In our example: assume a.f can call any of: `A.f()`, `B.f()`, `C.f()`, `D.f()`

Class Hierarchy Analysis: Example



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**

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 "A"; }
}
```

```
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 "A"; }
}
```

```
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 "A"; }
}
```

```
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 "A"; }
}
```

```
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 "A"; }
}
```

```
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 "A"; }
}
```

```
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());
    }
  }
}
```

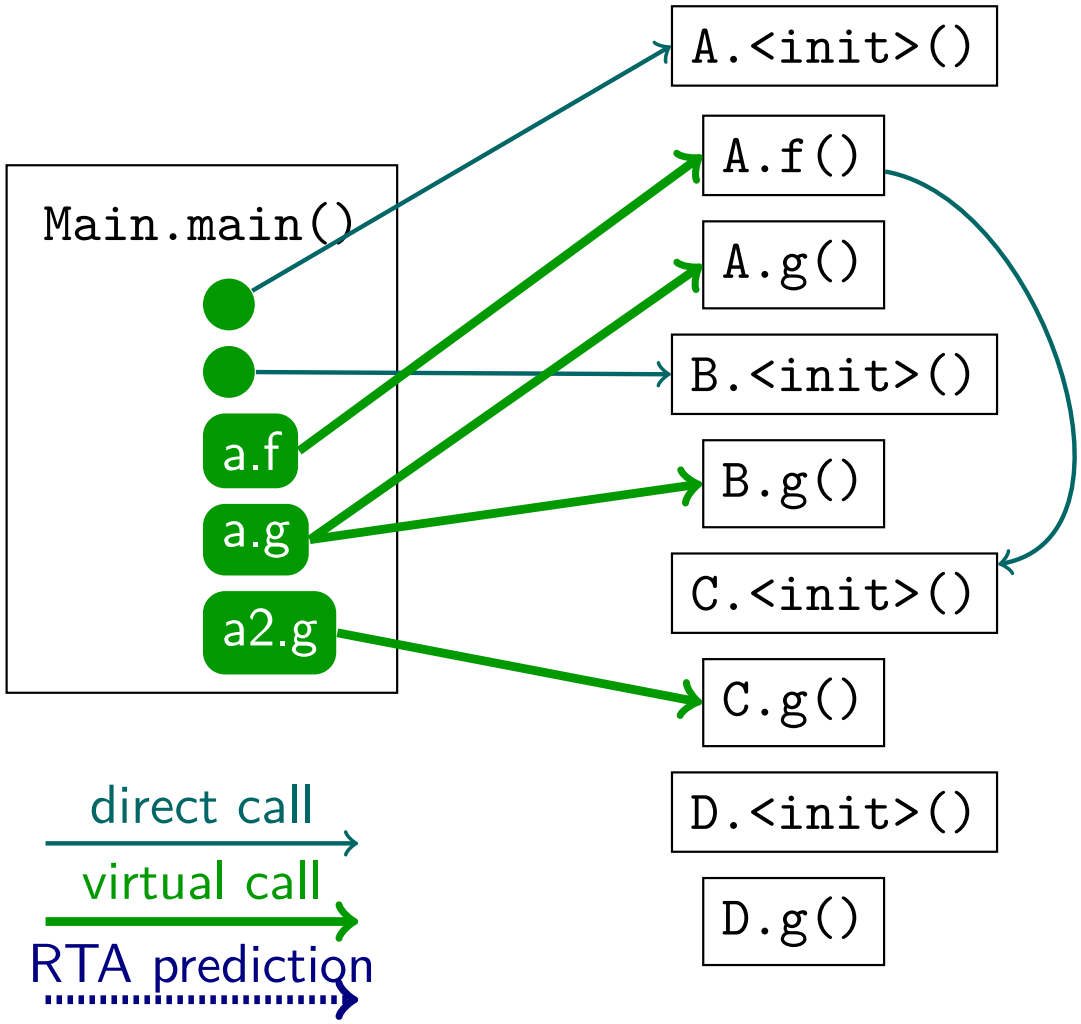
```
class A {
  public A
  f() { return new C(); }

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

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

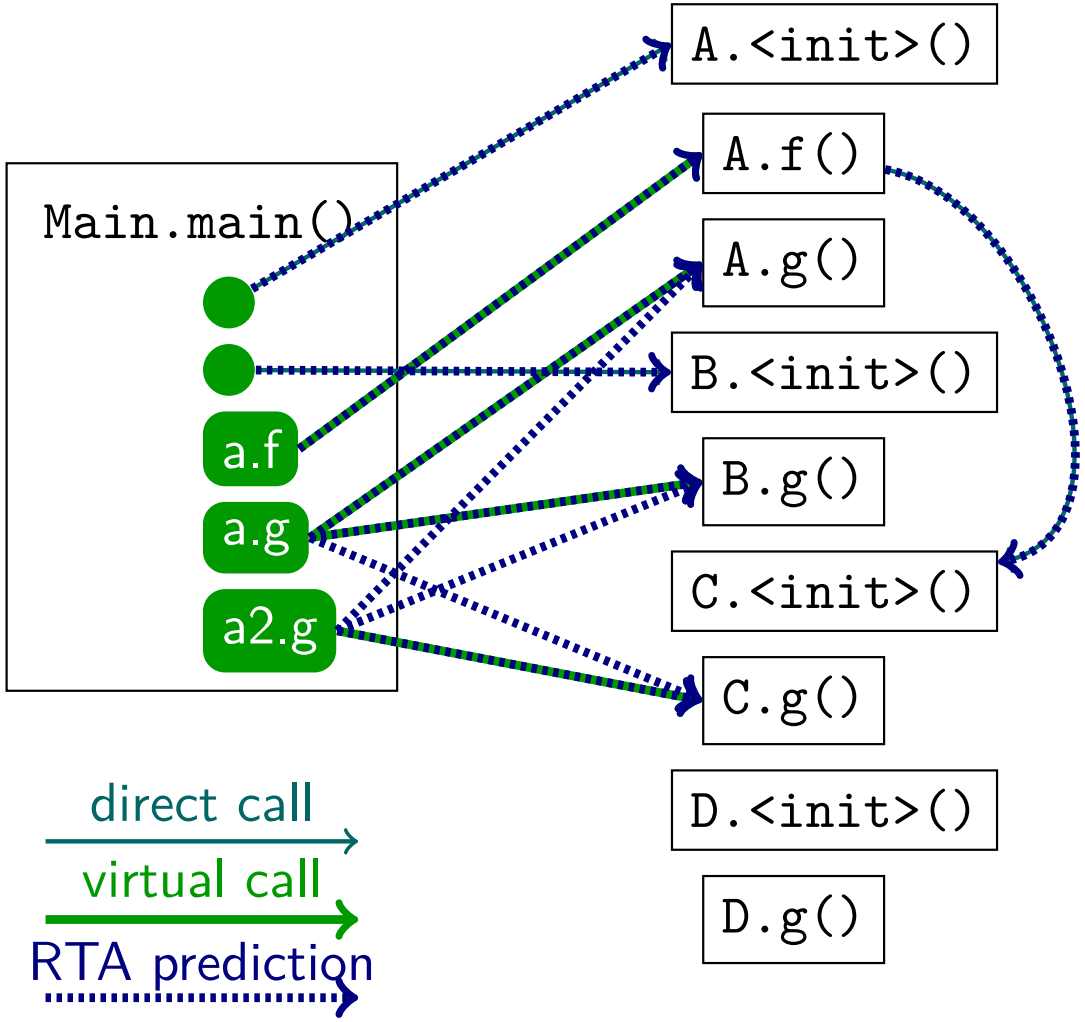
```
class C extends A {
  @Override
  public String
  g() { return "A"; }
}
```

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

Rapid Type Analysis: Example



Rapid Type Analysis: Example



Rapid Type Analysis Algorithm Sketch

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