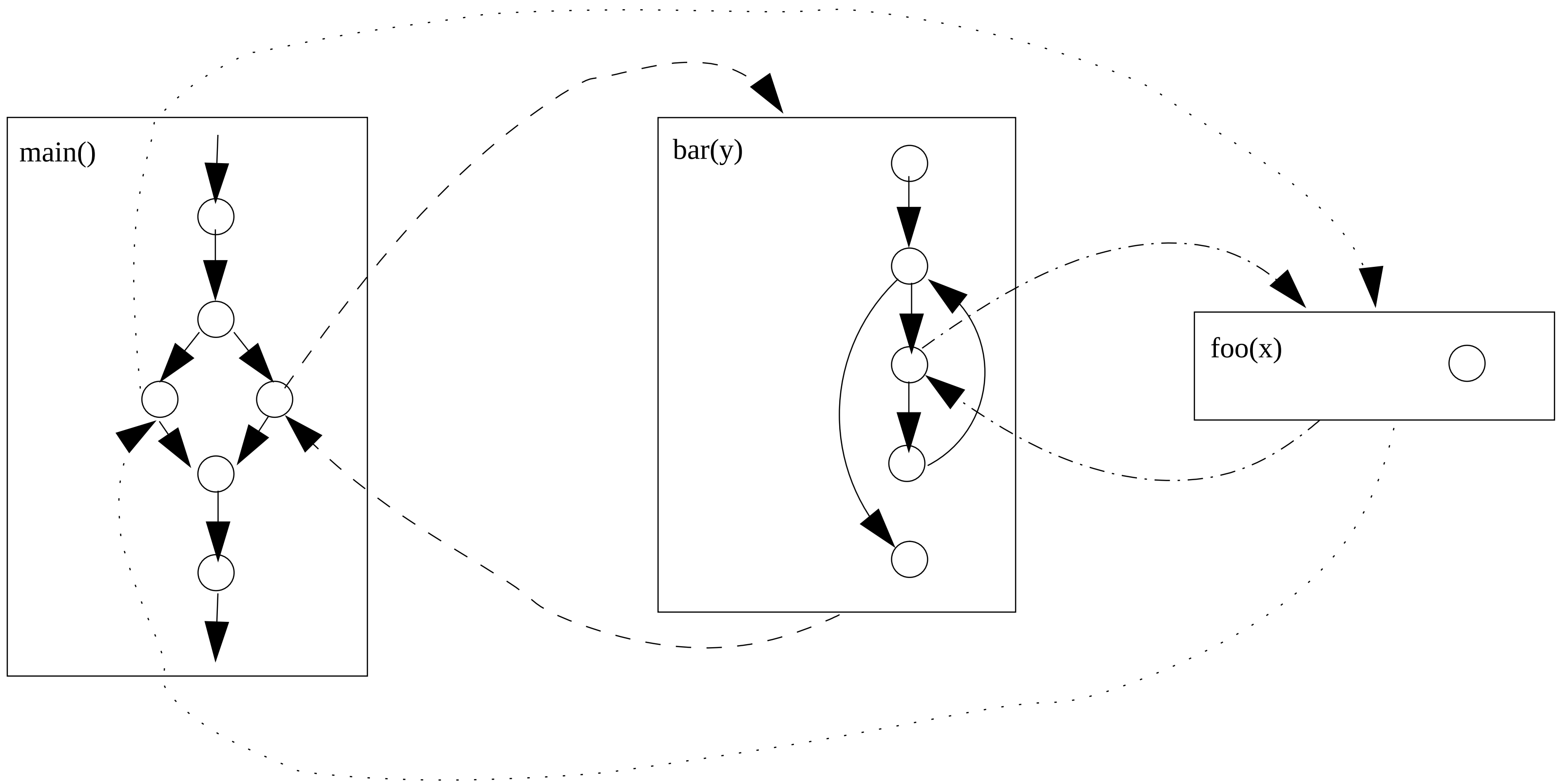
```
main() {
  ...
  if (x > 0) {
    x = foo(x);
  } else {
    x = bar(x);
  }
  ...
}
```

```
int bar(int y) {
  int t;
  while (y > 0) {
    t = t + foo(y);
    t++;
  }
  return t;
}
```

```
int foo(int x) {
  return x + x;
}
```

Context-sensitive approaches attempt to match up call sites with called methods and distinguish values flowing from different call sites.


main()
bar(y)
foo(x)

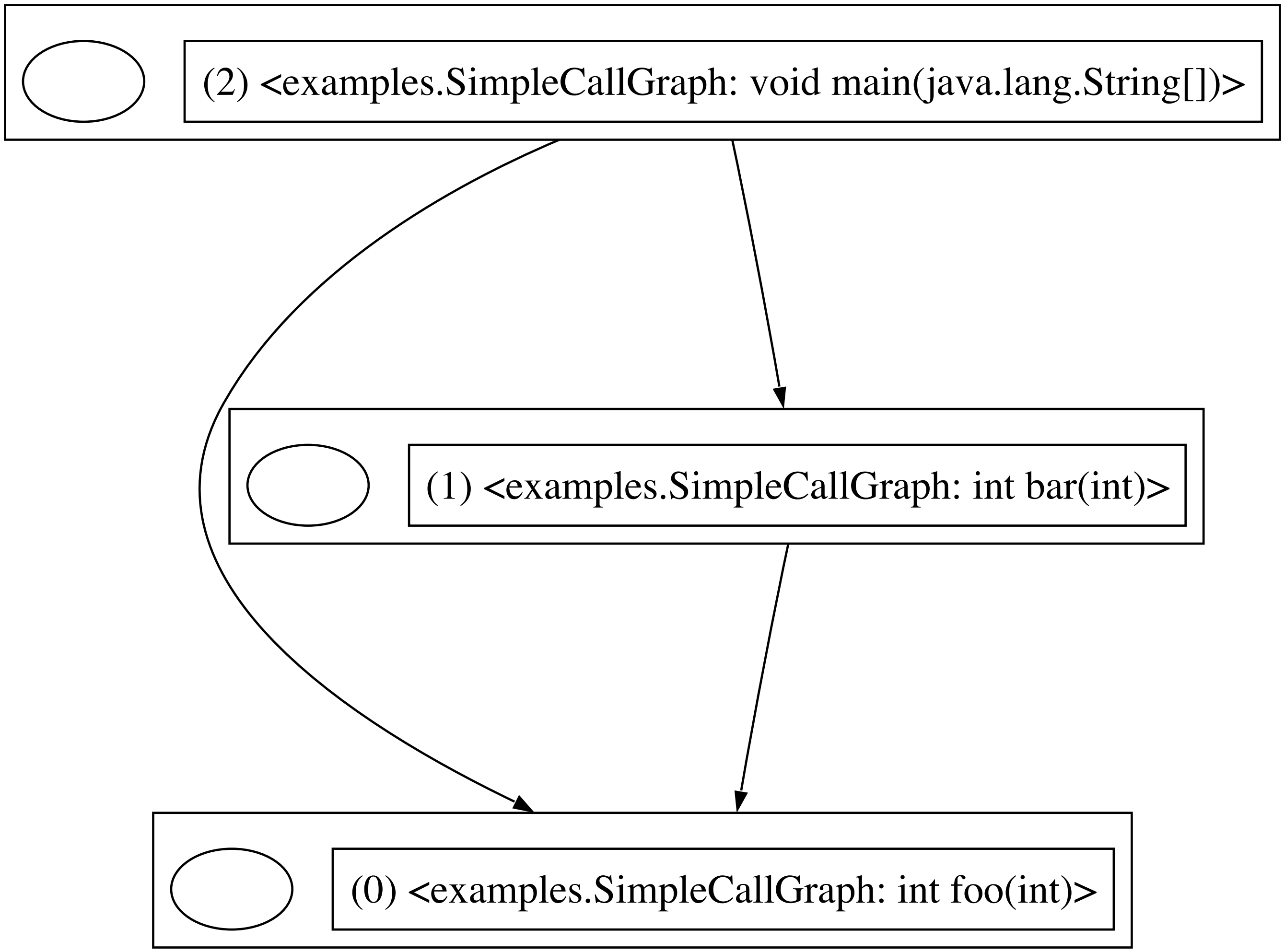
Call Graph

A *call graph* is an abstraction of the possible calling relationships among program methods

Just like a control flow graph a call graph *overestimates* the actual program behavior

It also leaves out alot of detail

- no information about the call sites in a method
- no information about the number of calls in a method
- no information about the order of calls in a method

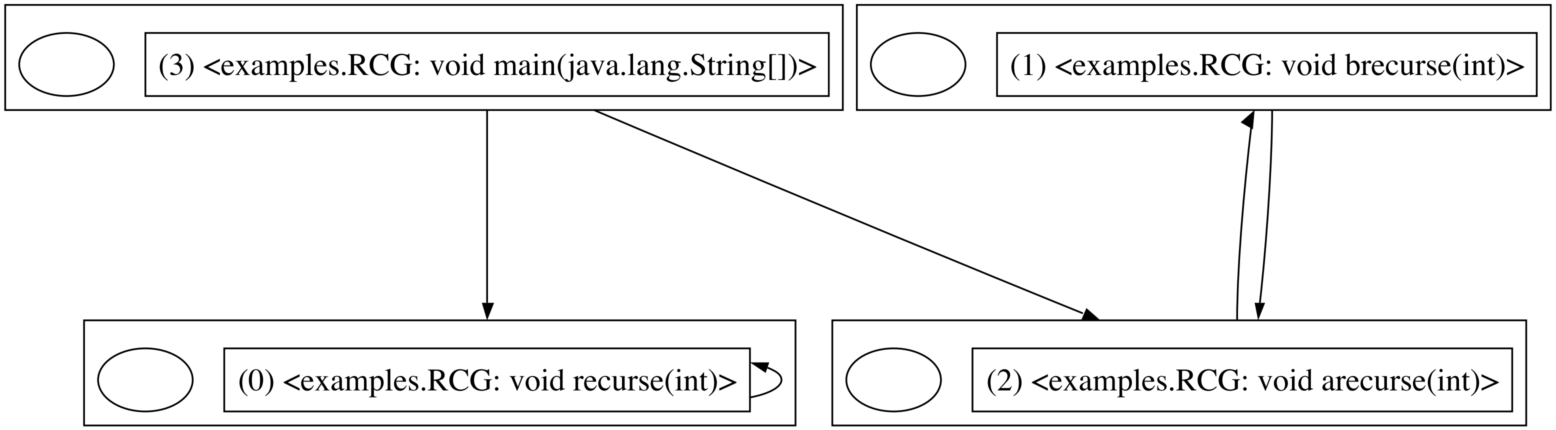


SummaryCallGraph

Call Graph : Recursion

```
void main() {
  arec(10);
  rec(7);
}

void rec(int x) {
  if (x > 0) rec(x--);
}

void arec(int x) {
  if (x > 0) brec(x--);
}

void brec(int x) {
  if (x > 0) arec(x--);
}
```

Need to be able to reflect both *direct* and *indirect* recursion.



SummaryCallGraph

Call Graph : Polymorphism

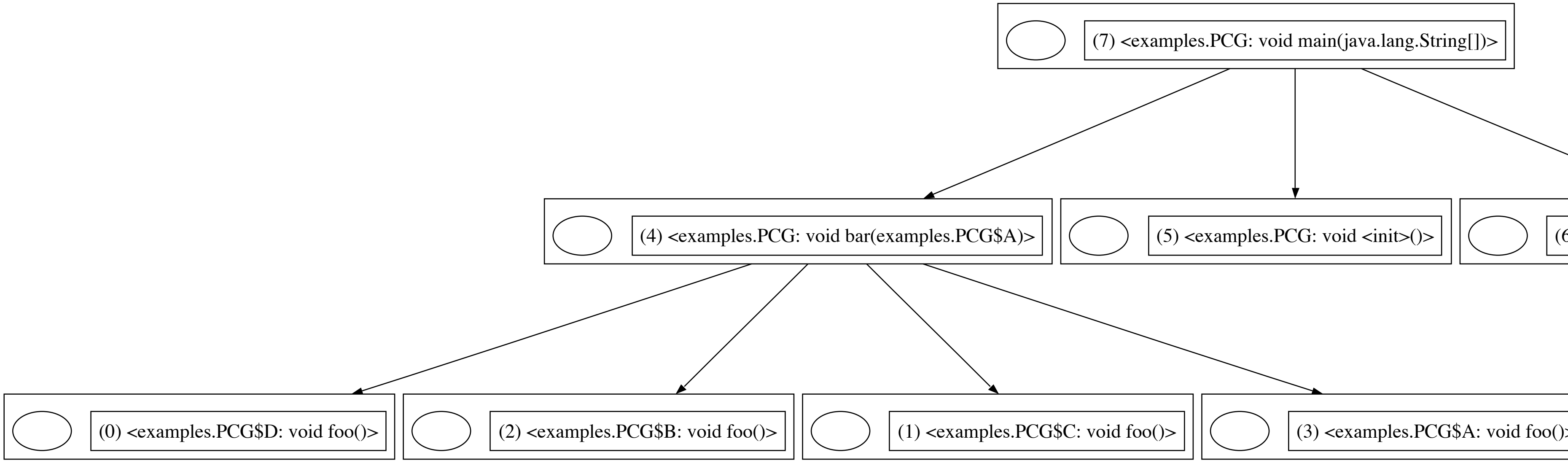
```
class PCG {
  class A { void foo() { x = 1;} }
  class B extends A { void foo() { x = 2; } }
  class C extends A { void foo() { x = 3; } }
  class D extends B { void foo() { x = 4; } }

  void main() { bar(new PCG().new A()); }

  static void bar(A a) { a.foo(); }
}
```

Dynamic dispatch can lead to significant overestimation of the possible calling relationships.

This is one reason *points to* analysis is so important.

Which calls to `foo` in `bar` are possible?

Interprocedural Analysis



SummaryCallGraph

Summary-based Interprocedural Analysis

The basic idea is to perform *dependent* but separate local flow analyses for each method

Dependences between methods, i.e., `foo()` calls `bar()`, are captured by constructing and applying *method summaries*

Summaries are calculated during the analysis until a fix-point is reached

Just as in a local flow analysis order is important, so an interprocedural analysis orders the calculation of summaries according to call dependences, i.e., analyze methods in reverse order of calls.

Method Summaries

A method summary reflects the data flow values that are computed on output of a method call

i.e., the least-upper bound of out at `return` stmts

Summaries are often elements of the underlying flow analysis lattice D, but they need not be.

If a method m calls another method n, the values on exit of m may depend on the summary for n. If the summary for n changes we want force a recompution of m's summary.

Applying Method Summaries

When analyzing a method body we use a summary to calculate a *method call transfer function* for a call to n

- if n already has a computed summary we use it
- if n has no stored summary we use the *default* summary

Since summaries account for all of the behavior of a method, transitively through all of its possible calls, a summary can be quite imprecise.

Methods that are not subjected to analysis, e.g., library methods, are assigned a *default* summary.

Parameterized (or Context-sensitive) Method Summaries

One can construct summaries that are functions $D \to \mathcal{P}(D)$

this can be generalized to some other context Δ as the domain

Intuitively, the domain value defines a calling context and the image defines the summarized effects of the methods in that context.

To construct such a summary, for each value $d \in D$ repeat the following

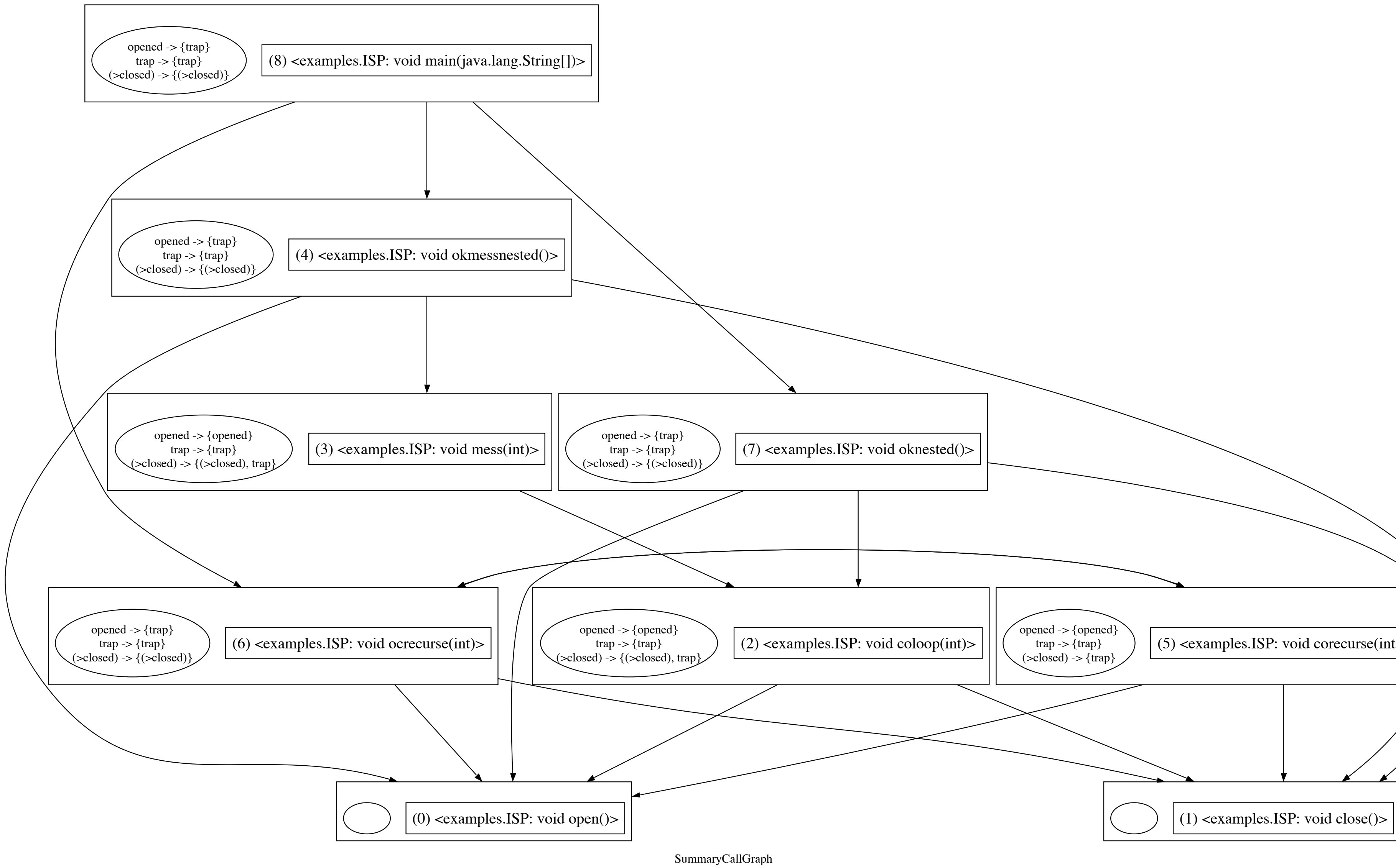
1. set $\iota = d$ for the extremal node
2. run flow analysis to fix-point
3. calculate $s = \bigsqcup_{r \in Returns} A_{out}(r)$
4. install the map entry $[d \mapsto s]$ in the summary

Interprocedural State Propagation Analysis

```
void oknested() {
 open(); coloop(); close();
}

void coloop(int x) {
 while (x < 10) {
  close(); x++; open();
 }
}

void okmessnested() {
 open(); mess(); close();
}
```

```
void ocrecurse(int x) {
 open();
 if (x > 0) corecurse(x--);
}

void corecurse(int x) {
 close();
 if (x > 0) ocrecurse(x--);
}

void mess(int x) {
 if (x > 0)
  while (x-- > 0) coloop(x);
 else coloop(x); }
```

Interprocedural Analysis



Order of Analysis

Bottom-up in the call graph (but there is lots of freedom to break tie).

The first ones to process are the *leaves*: `open()`, `close()`

`coloop()` is called from multiple methods so we need to process this relatively early in the order.

The chain of `mess()`, `okmessnested()` are processed in reverse call order.

This cluster is *co-dependent* we actually have to reanalyze one of them: `corecurse()`, `ocrecurse()`, `corecurse()`

`oknested()` could have come earlier, but need not

and finally `main()`