Compiler Analyses for Improved Return Value Prediction

Christopher J.F. Pickett Clark Verbrugge

{cpicke,clump}@sable.mcgill.ca

School of Computer Science, McGill University

Montréal, Québec, Canada H3A 2A7

Overview

- Introduction and Related Work
- Contributions
- Framework
- Parameter Dependence Analysis
- Return Value Use Analysis
- Conclusions
- Future Work

Introduction and Related Work

- Speculative method-level parallelism (SMLP) allows for dynamic parallelisation of single-threaded programs
  - speculative threads are forked at callsites
  - suitable for Java virtual machines
- Perfect return value prediction can double performance of SMLP (Hu *et al.*, 2003)
- Implemented accurate return value prediction in SableVM, our group's JVM (Pickett *et al.*, 2004)
- Current goals:
  - Reduce memory requirements
  - Achieve higher accuracy

Speculative Method-Level Parallelism

```
// execute foo non-speculatively
r = foo (a, b, c);

// execute past return point
// speculatively in parallel with foo()
if (r > 10)
   {
      ...
   }
else
   {
      ...
   }
```

Impact of Return Value Prediction

RVP strategy	return value	SMLP speedup
none	arbitrary	1.52
best	predicted	1.92
perfect	correct	2.76

- 26% speedup over no RVP with Hu's best predictor
- 82% speedup over no RVP with perfect prediction
  - Improved hybrid accuracy is highly desirable

- S. Hu., R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 5:1–21, Nov. 2003.

Return Value Prediction in SableVM

- Implemented all of Hu *et al.*'s predictors in SableVM
- Introduced new memoization predictor into hybrid

	hash(a,b,c)	return value
foo(7,5,3) →		11
foo(4,6,8) →		9
foo(9,1,2) →		10

- C.J.F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. *Second Value-Prediction and Value-Based Optimization Workshop (VPW2) at ASPLOS*, Boston, Massachusetts, Oct. 2004.

Return Value Prediction in SableVM

- Achieved 72% accuracy over SPEC JVM98
  - 81% if memoization is included
- But ...
  - Large amounts of memory are required
  - Still room for greater accuracy

- C.J.F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. *Second Value-Prediction and Value-Based Optimization Workshop (VPW2) at ASPLOS*, Boston, Massachusetts, Oct. 2004.

Contributions

- Static analyses in Soot
- Parameter dependence analysis
  - Eliminate unnecessary memoization inputs
  - Save memory
  - Increase accuracy
- Return value use analysis
  - Allow for use of incorrect predictions
  - Increase accuracy
- Convey results to SableVM using attributes

Framework

- Soot: Java bytecode compiler framework
  - Spark: points-to analysis and callgraph
  - Jimple: typed, stackless, 3-address IR
  - Baf: streamlined representation of Java bytecode
  - Attribute generation framework
- SableVM: portable Java virtual machine
  - Attribute parsing
  - Previous RVP implementation
- SPEC Client JVM98 Benchmark Suite
  - S100 (size 100), no harness
  - All benchmarks except `raytrace`

Parameter Dependence Analysis

- Memoization predictor
  - Hash together method arguments
  - One predictor hashtable per callsite

- Problem: redundant entries in hashtables

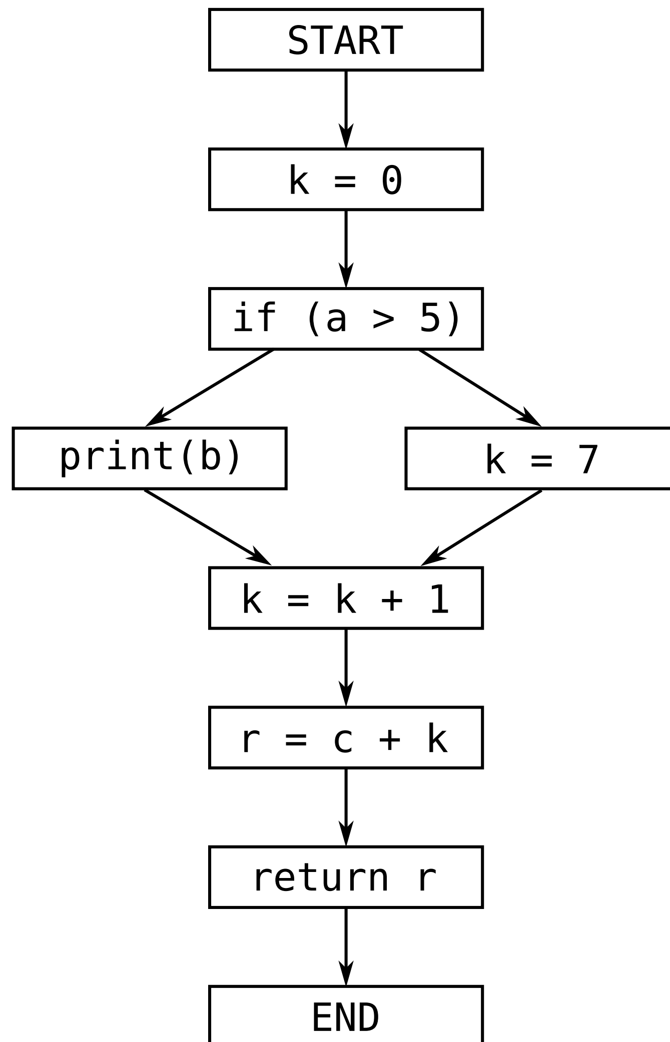
Parameter Dependence Analysis

- Insight: not all parameters affect return value
  - Eliminate inputs to predictor
  - Increase hashtable sharing
    - ++accuracy
    - −−size

hash(a,c)
return value
foo(7,5,3)
foo(7,2,3)
foo(7,8,3)
11

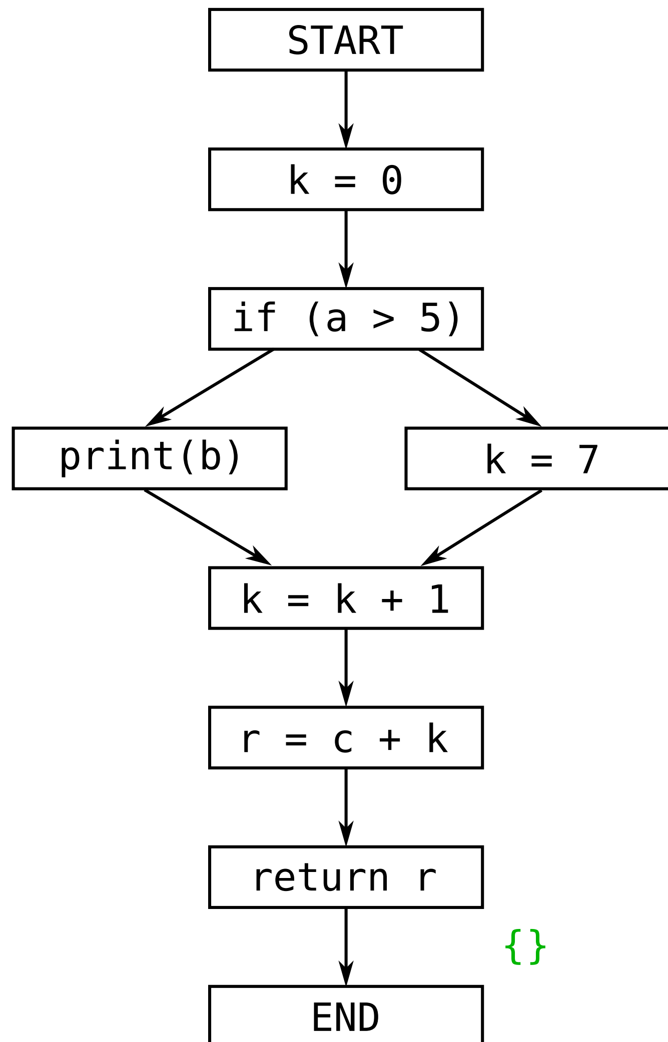
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

      int k, r
```



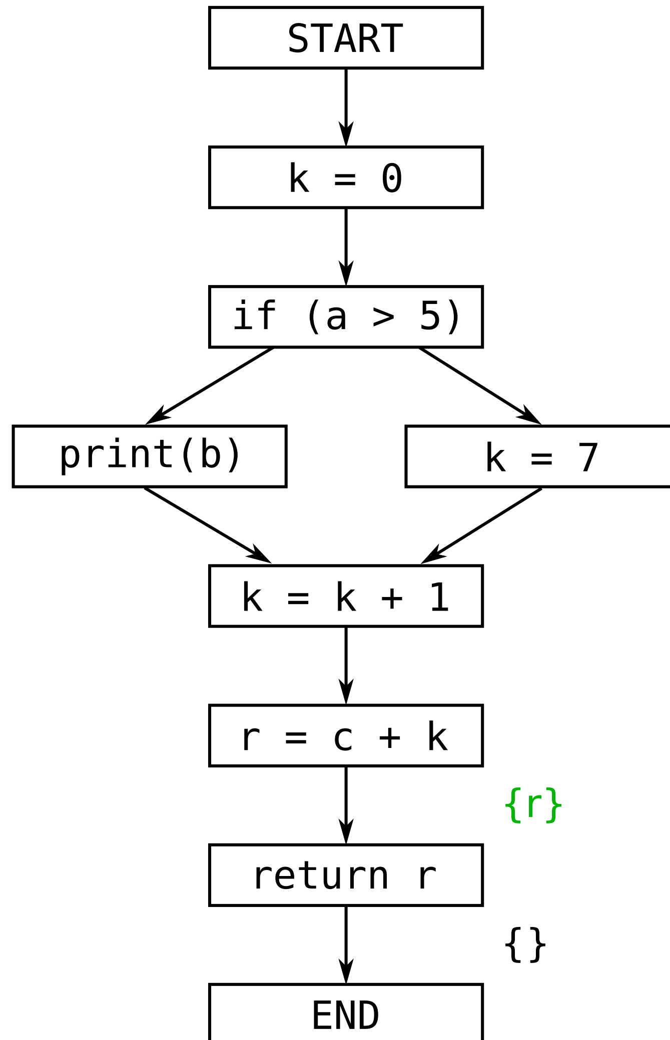
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

       int k, r
```



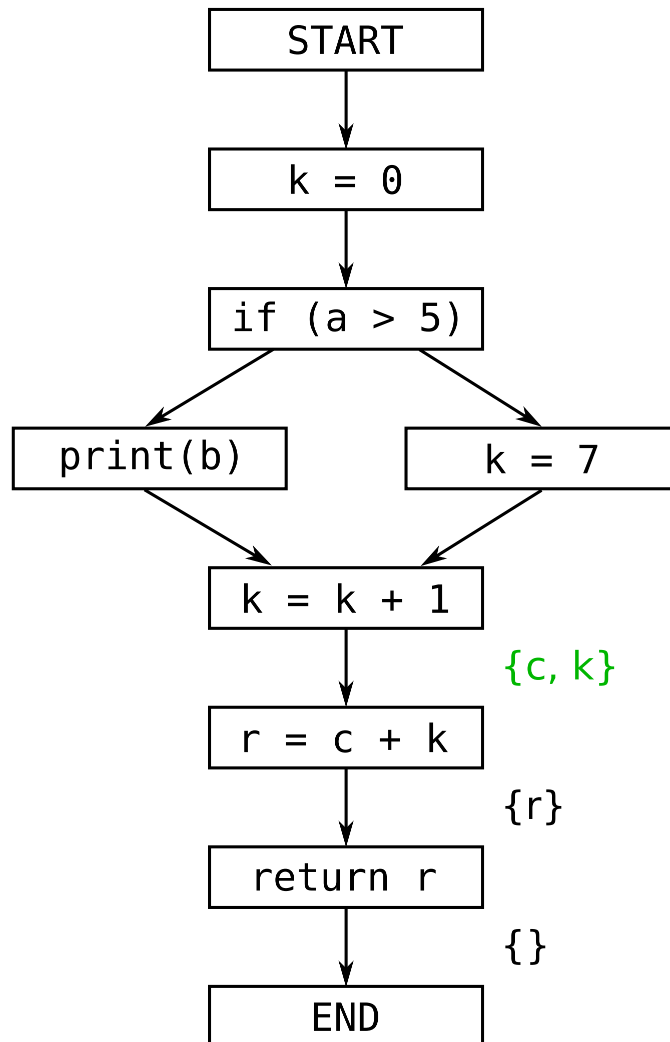
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

        int k, r
```

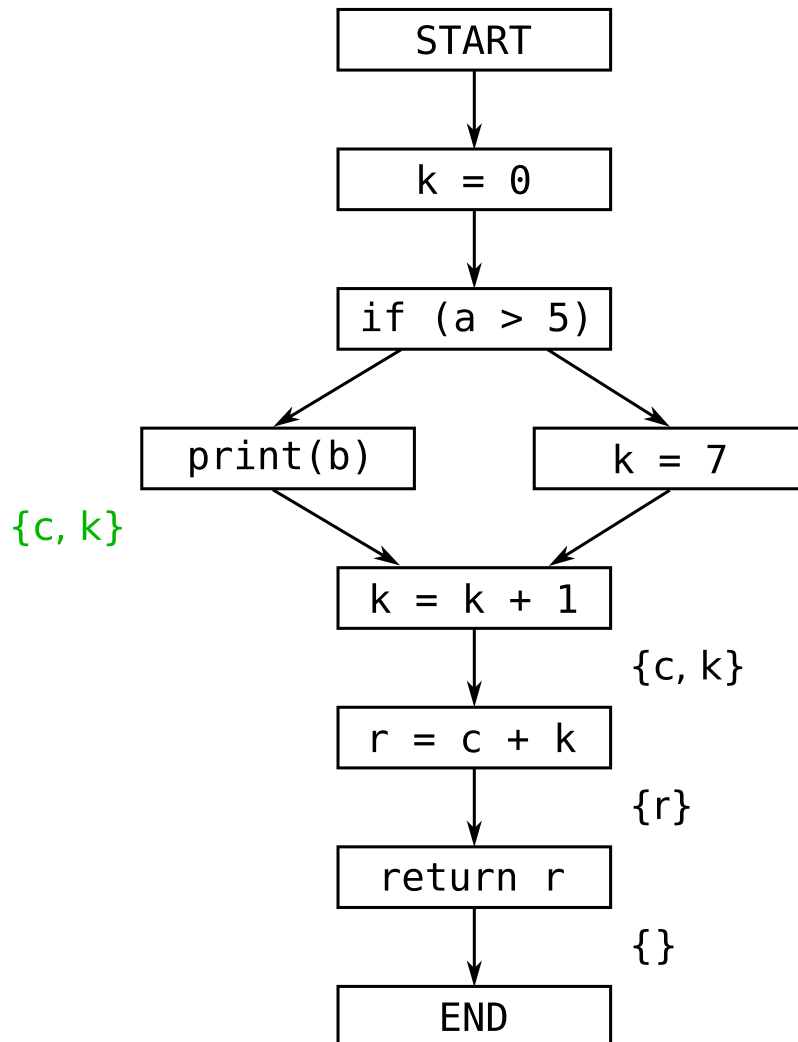


Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

        int k, r
```

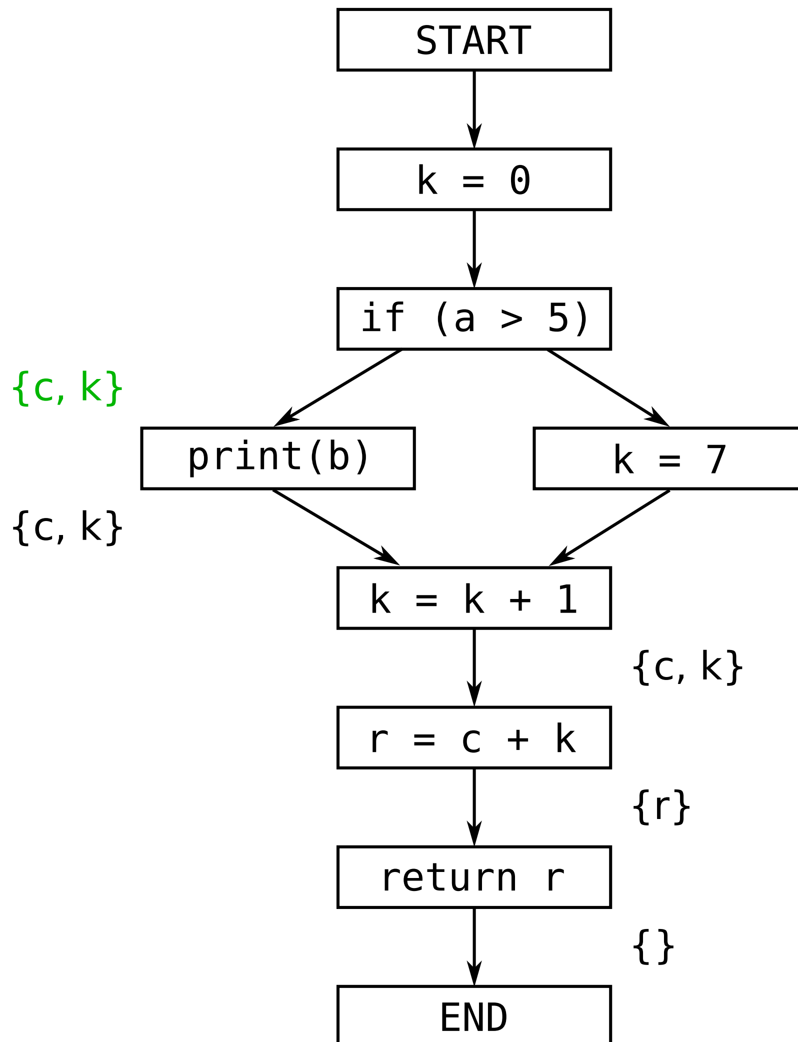


Intraprocedural Parameter Dependence



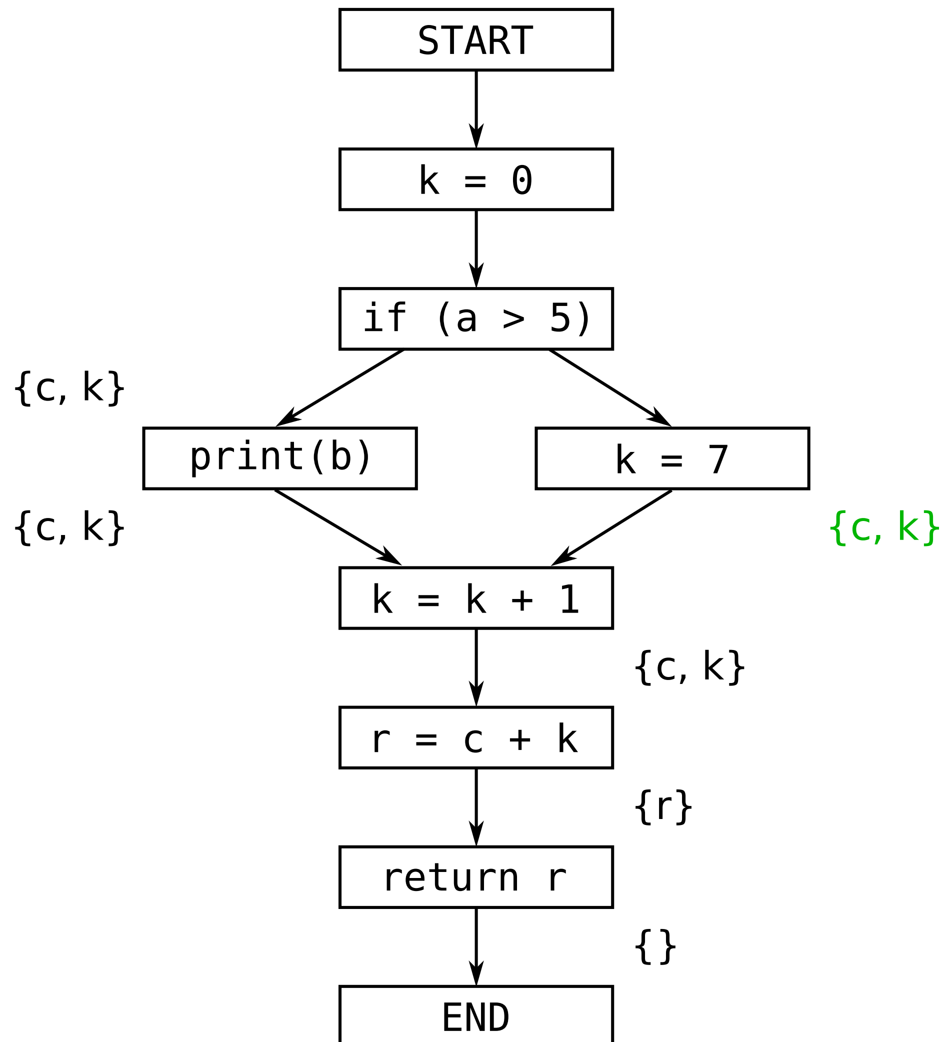
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

        int k, r
```



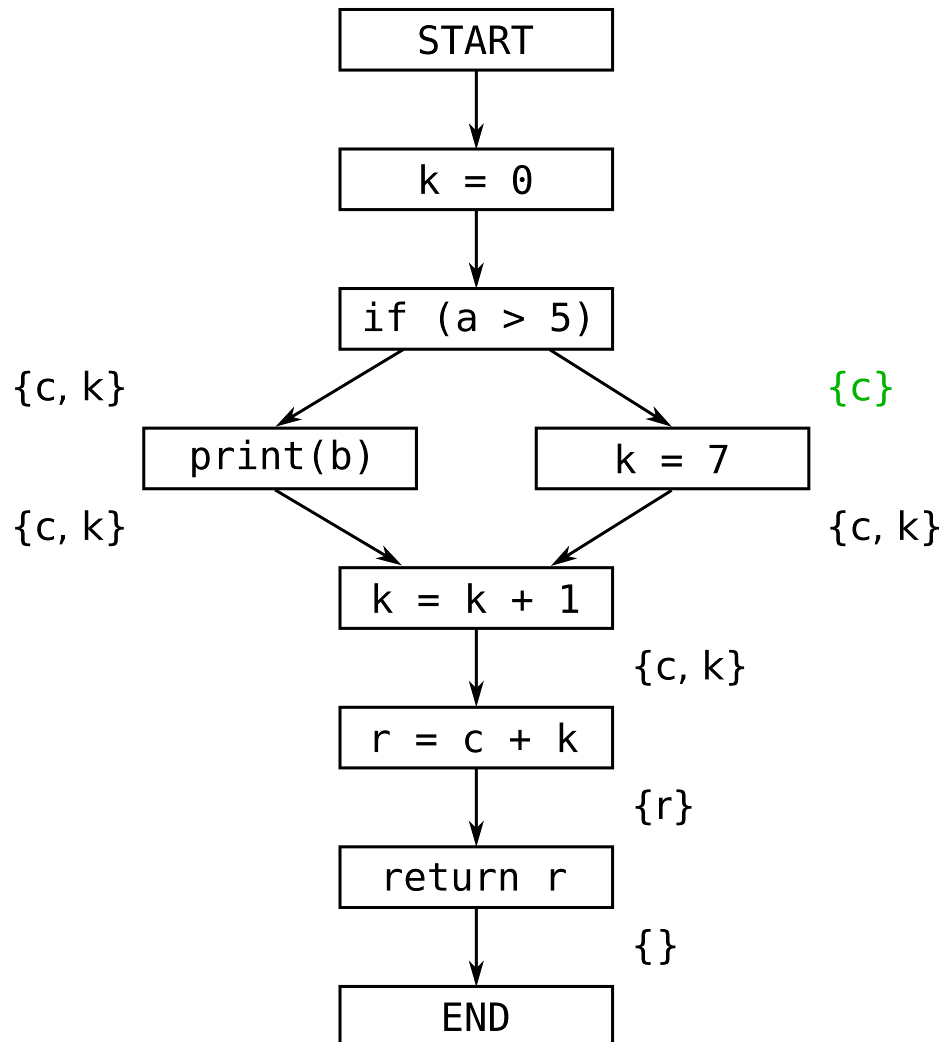
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

        int k, r
```



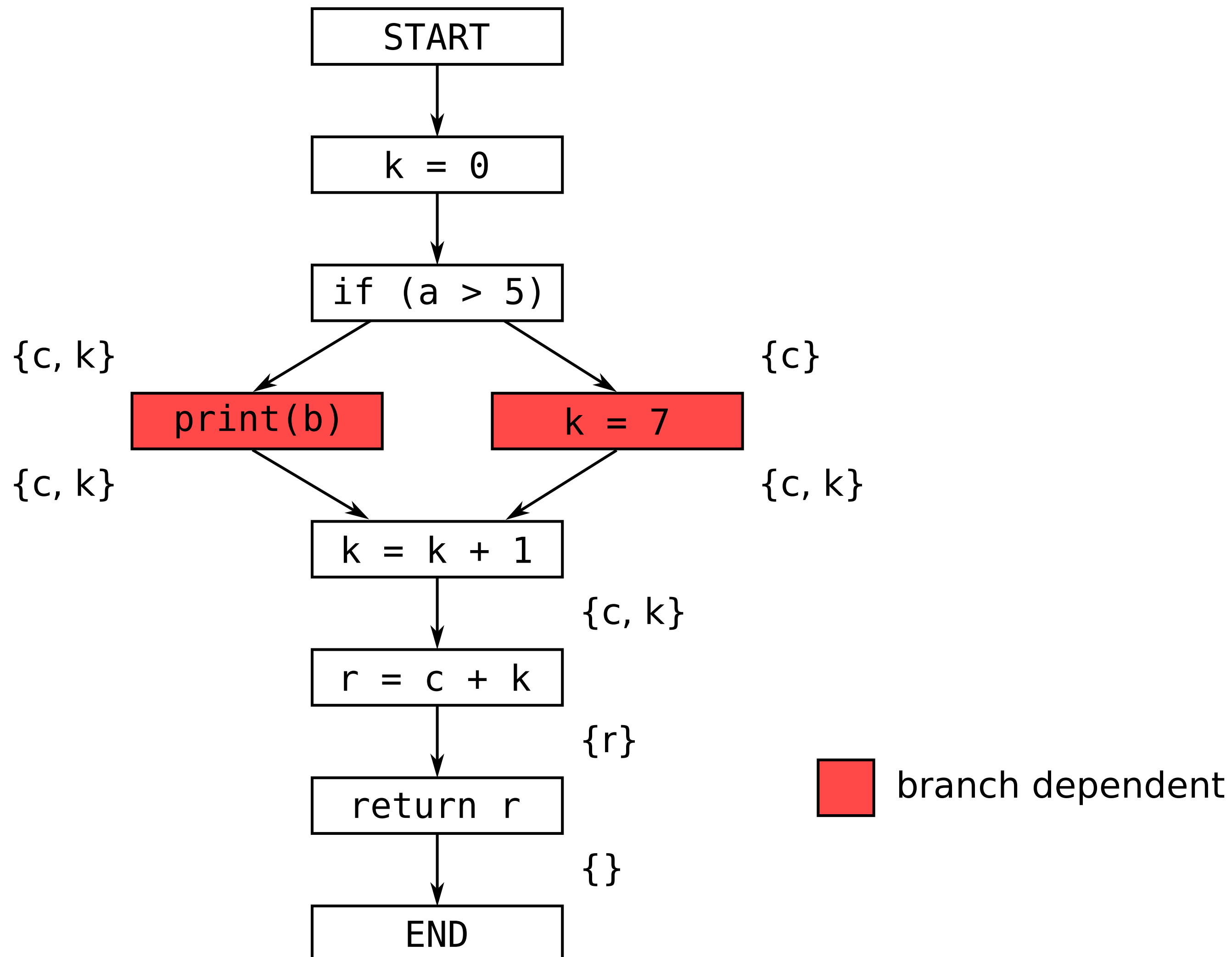
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

    int k, r
```



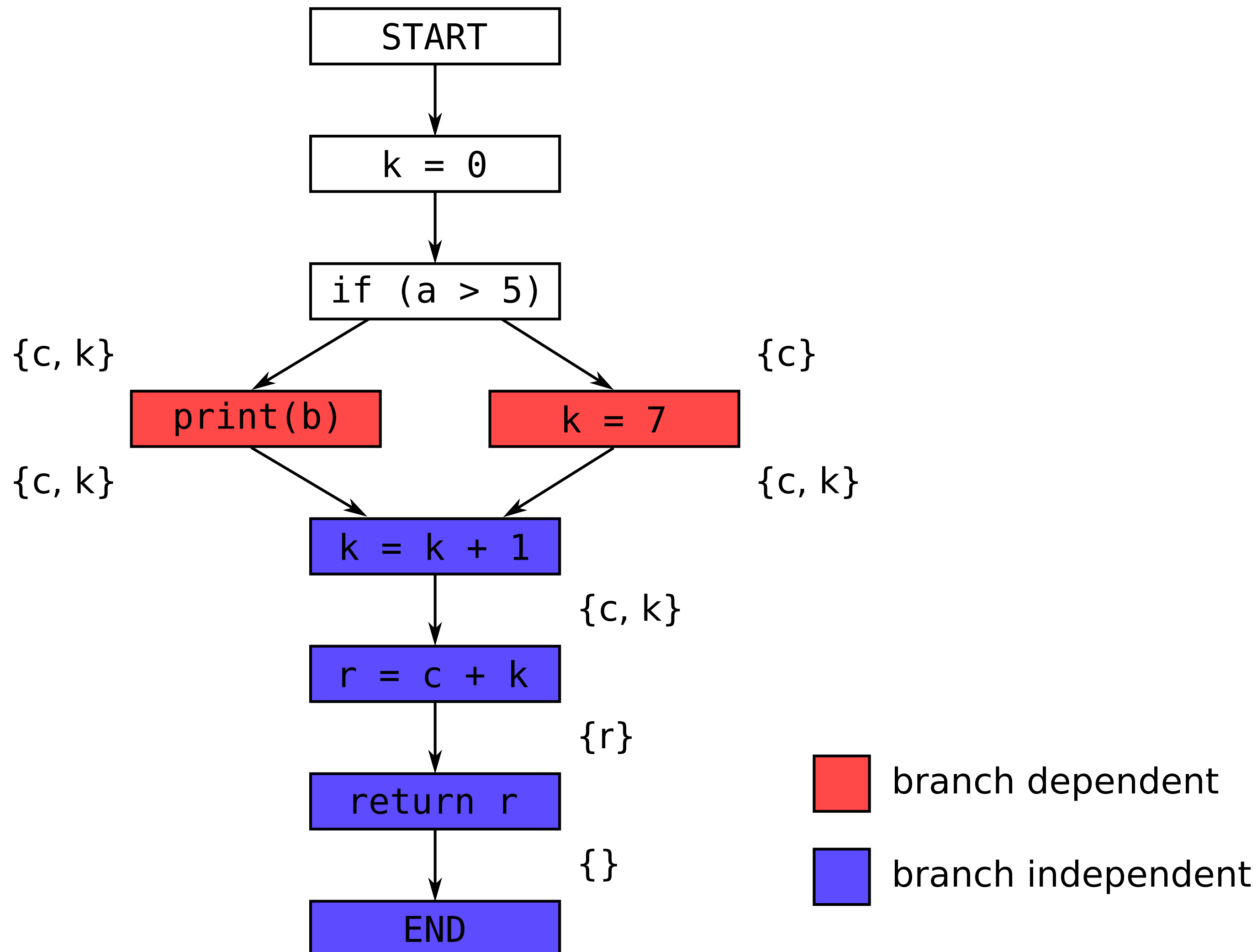
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

        int k, r
```

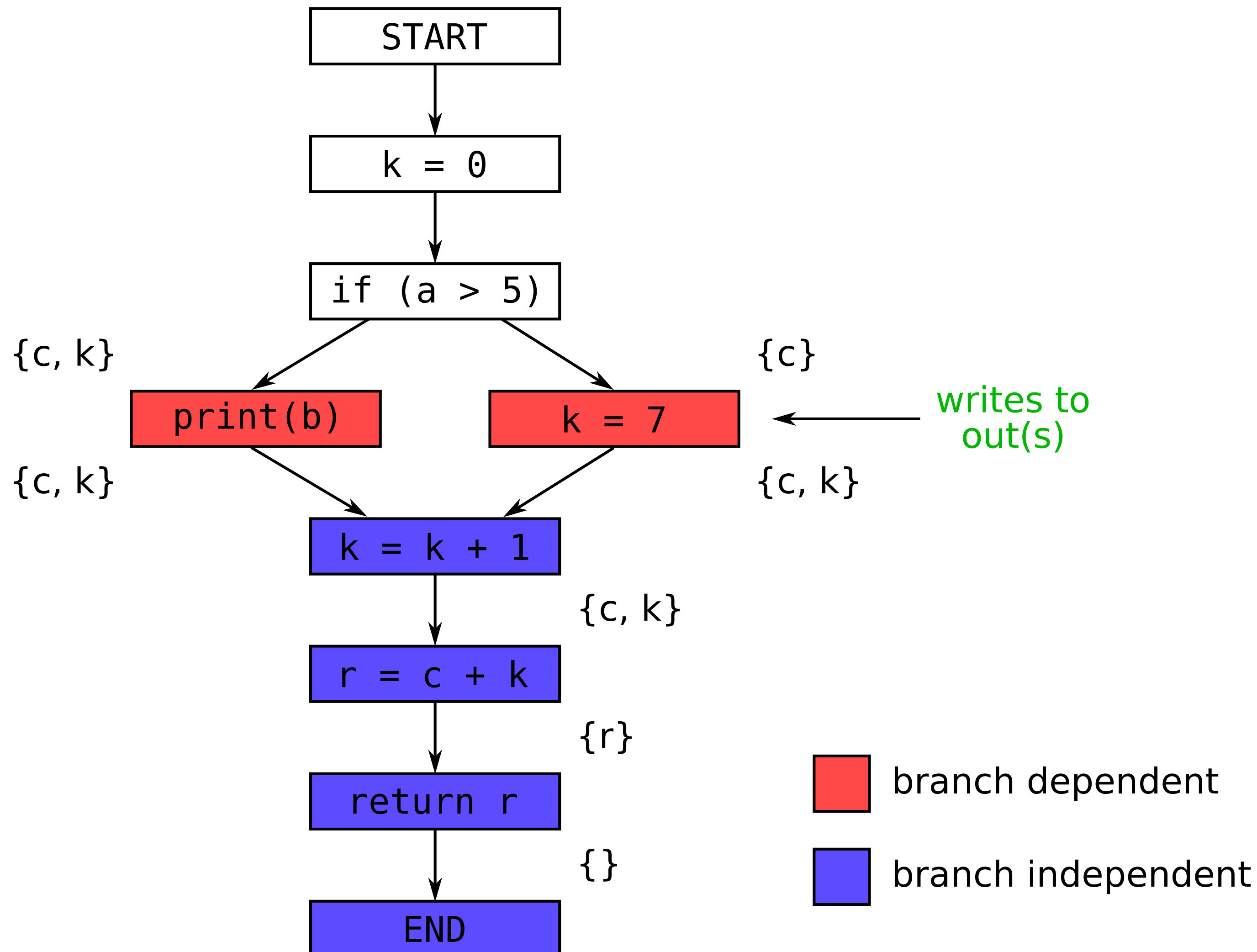


Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

        int k, r
```

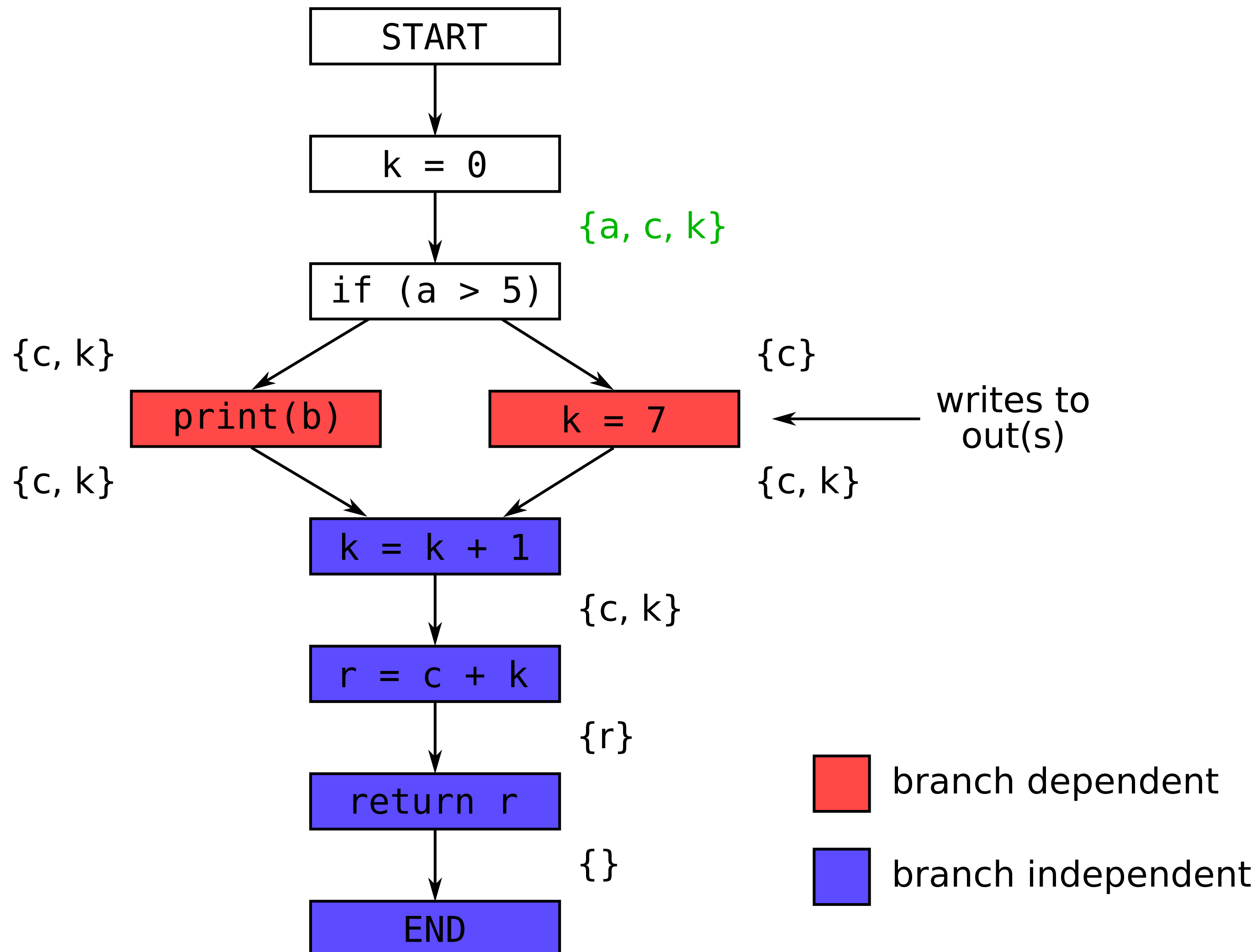


Intraprocedural Parameter Dependence

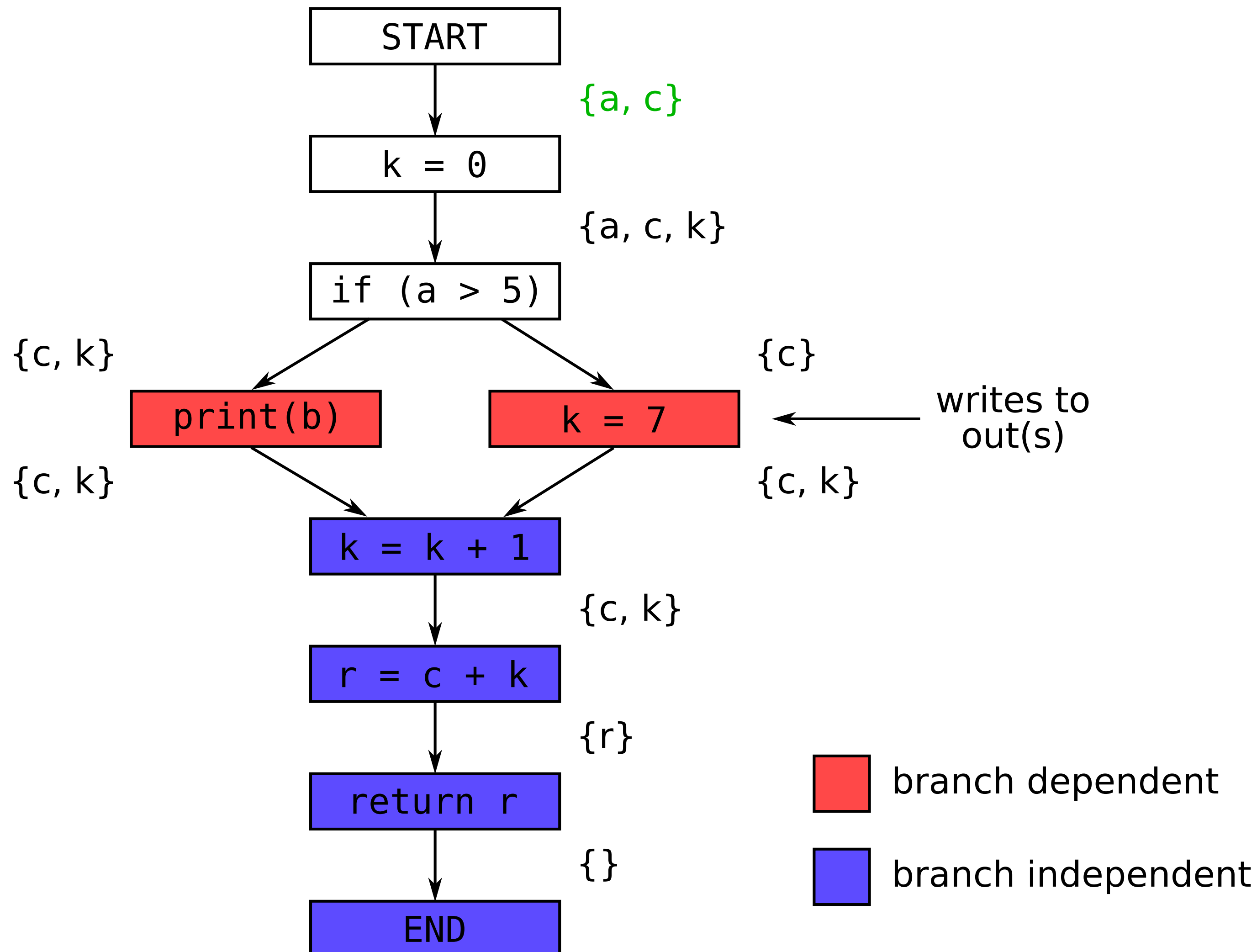


Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

       int k, r
```

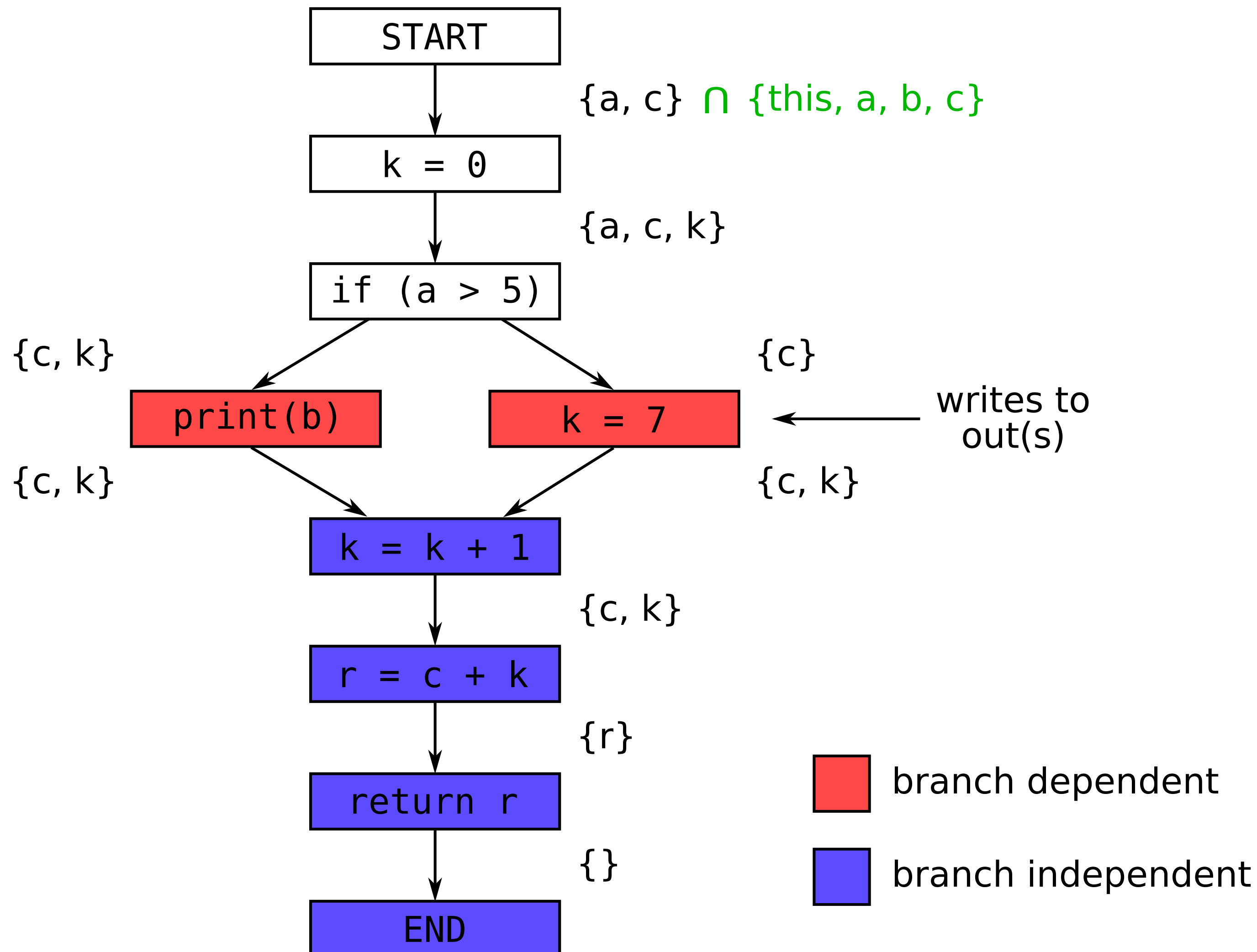


Intraprocedural Parameter Dependence



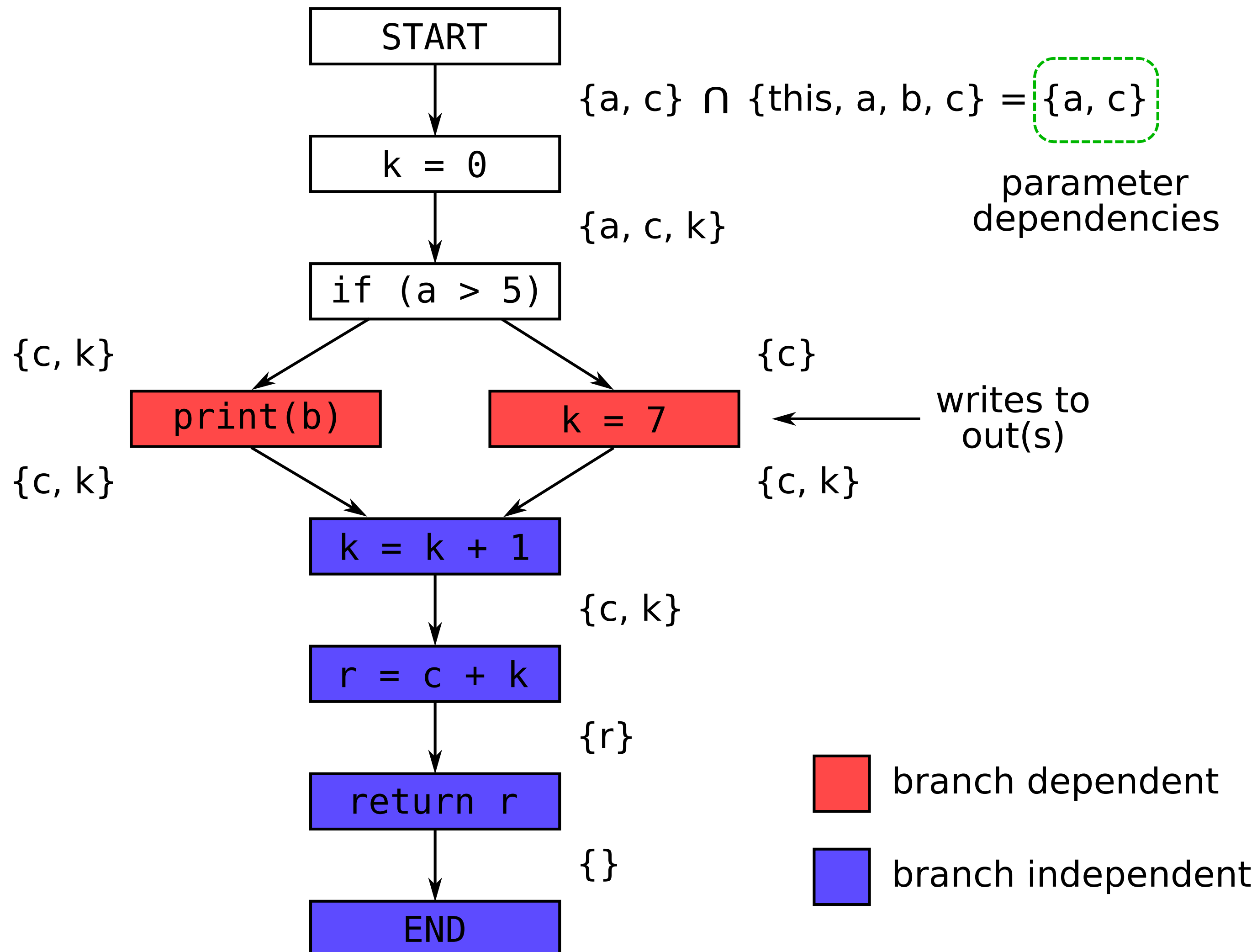
Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

    int k, r
```



Intraprocedural Parameter Dependence

```
public int foo (int a, int b, int c)

       int k, r
```



Interprocedural Parameter Dependence

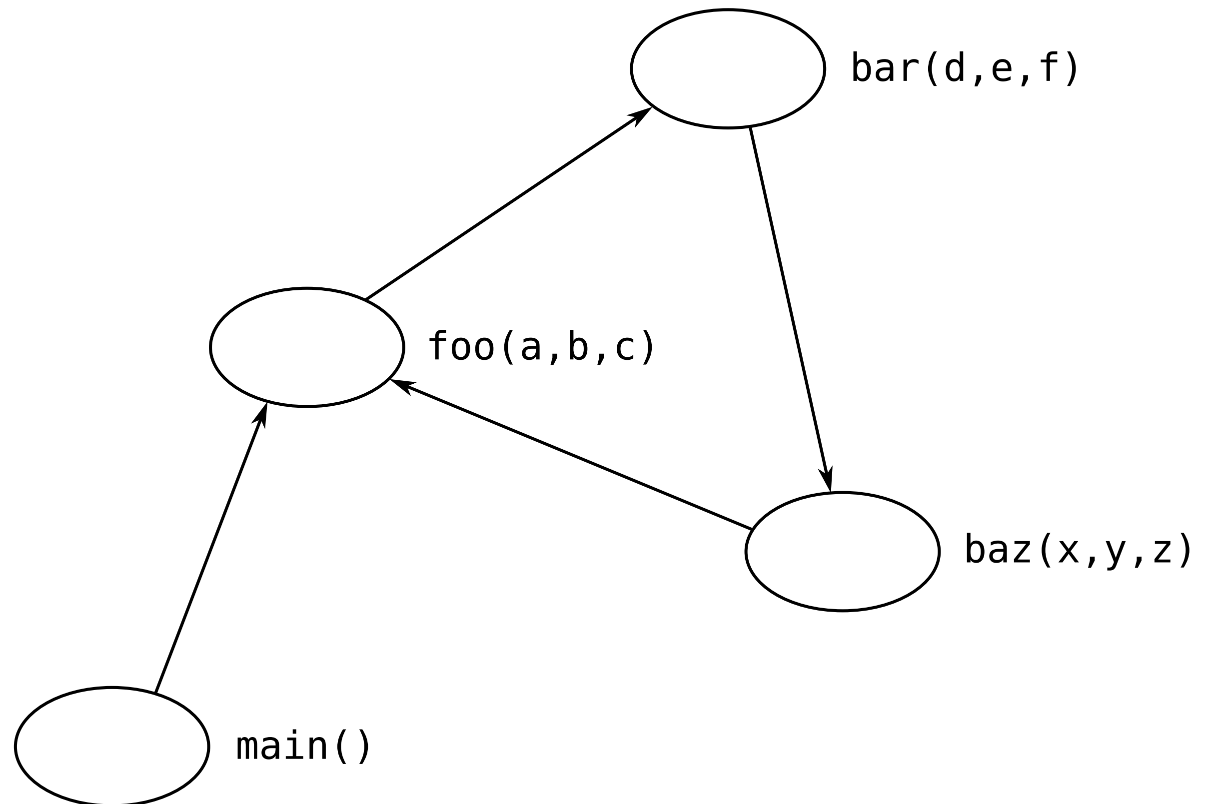
```
r = foo(a,b,c)
```

which uses do we add?

Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?



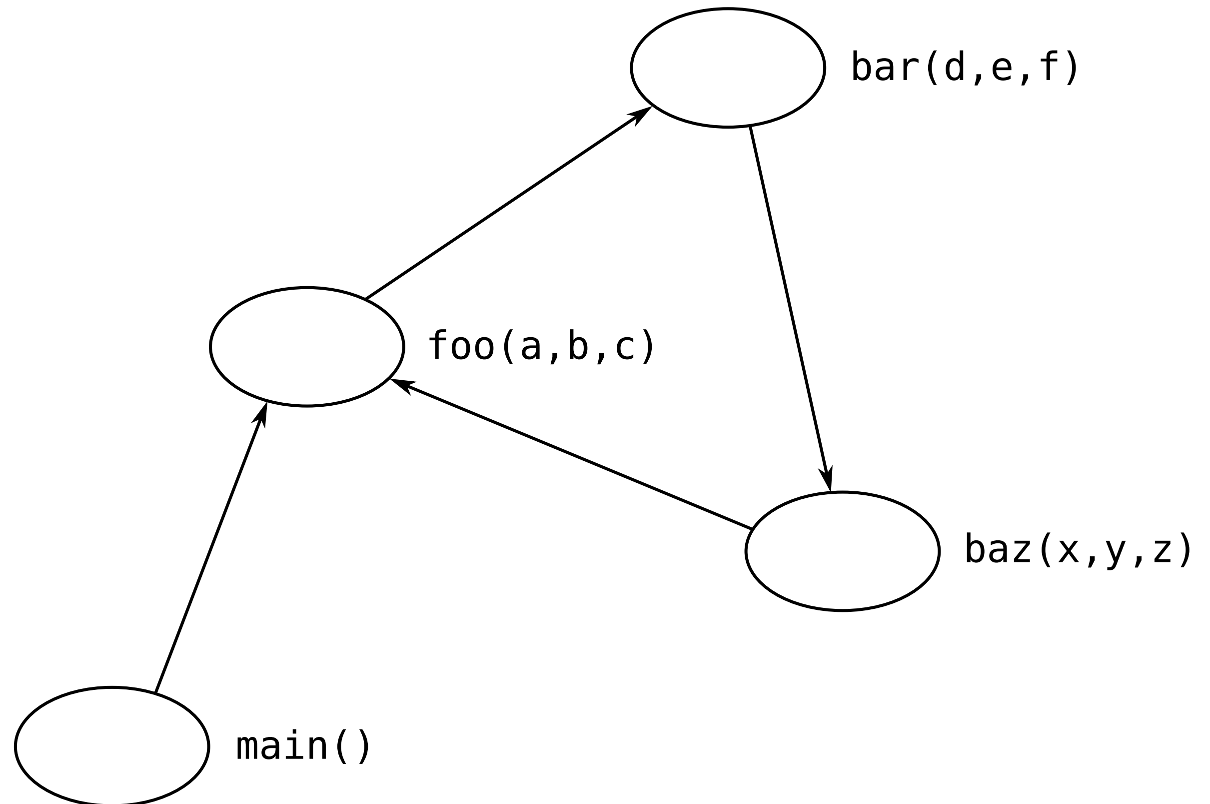
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

```
baz
bar
foo
main
```



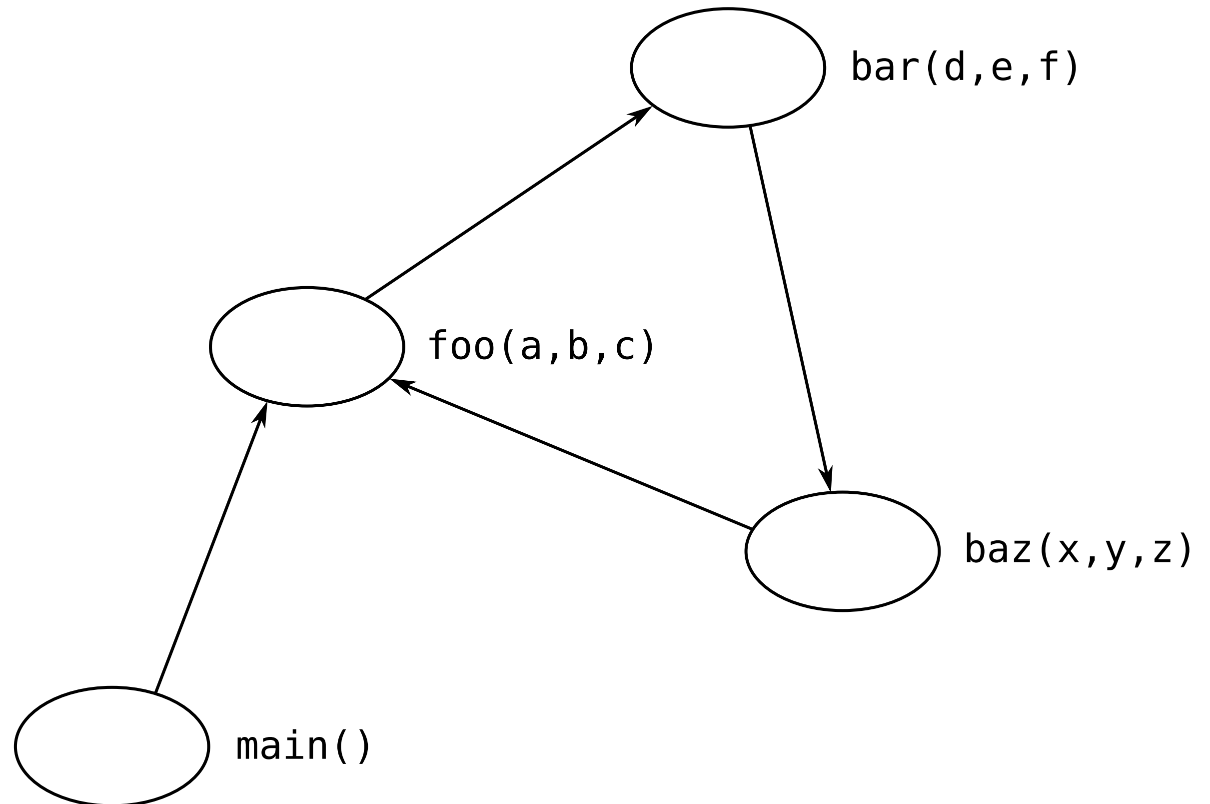
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

- ~~baz~~
- bar
- foo
- main



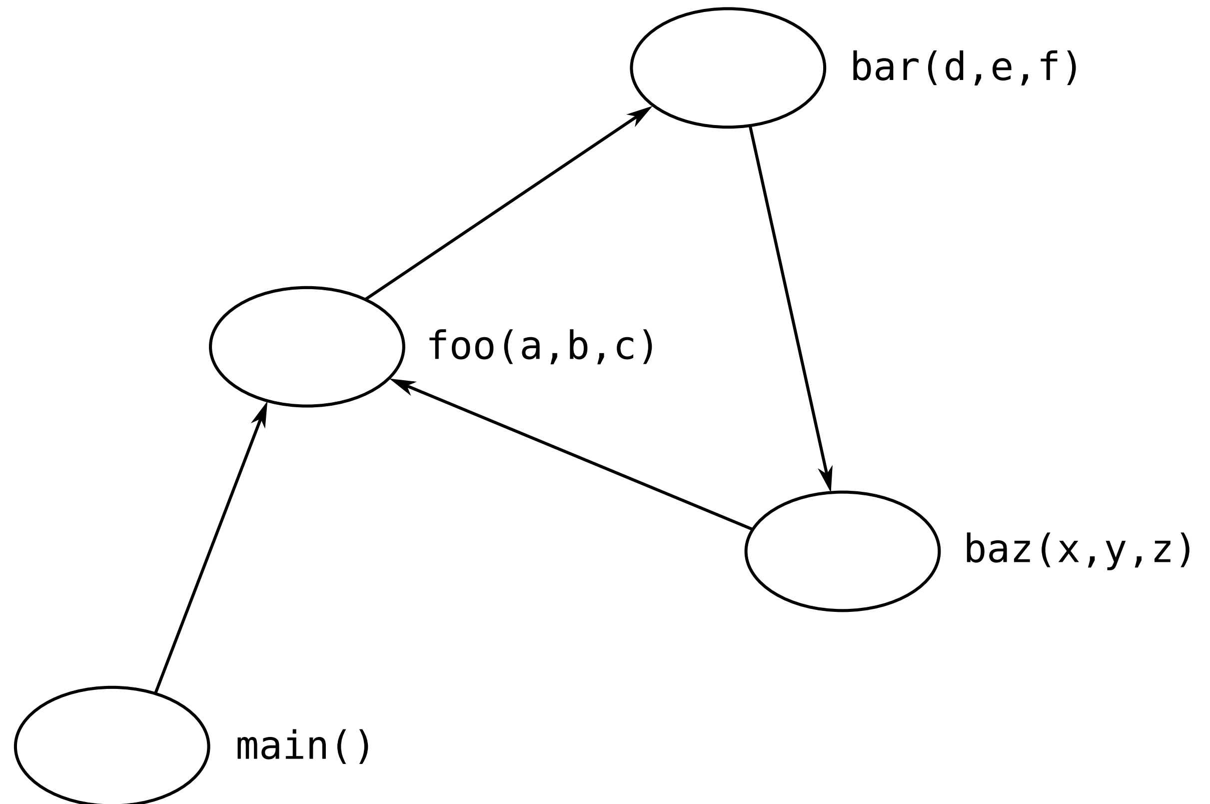
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

~~baz~~
~~bar~~
foo
main



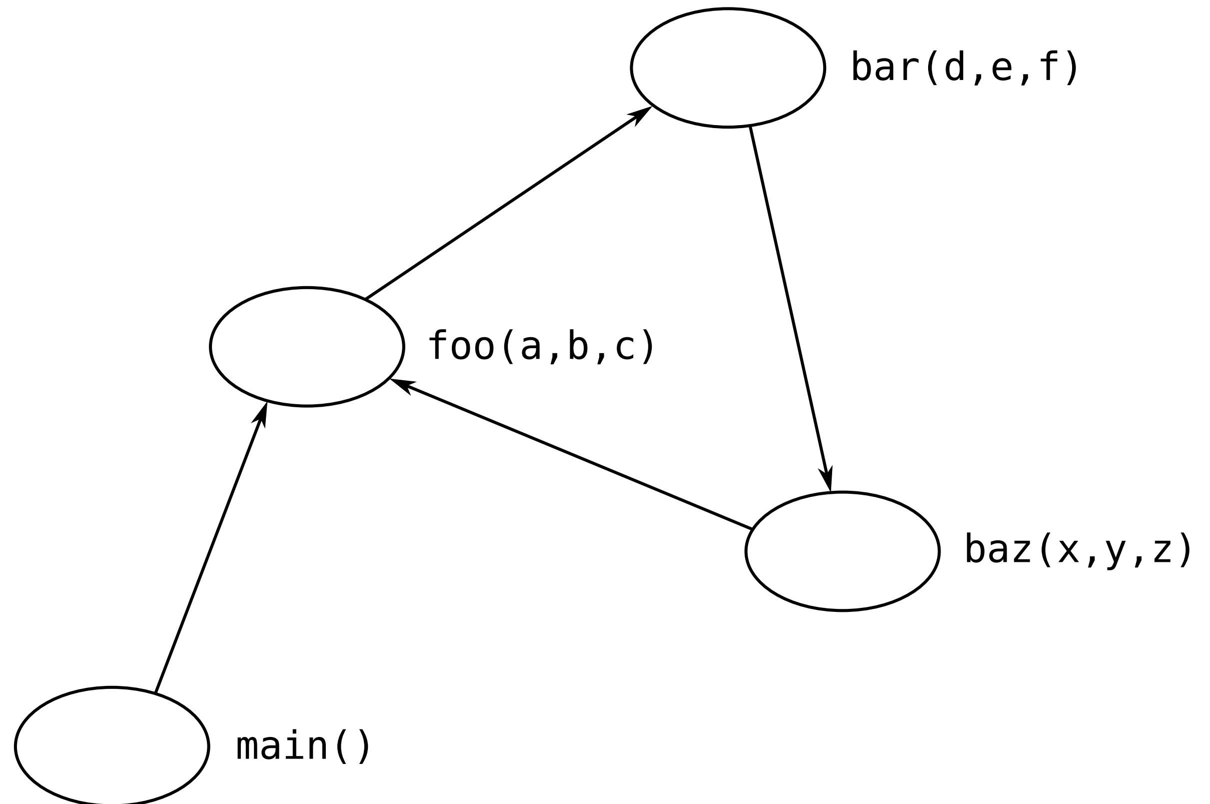
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

~~baz~~
~~bar~~
~~foo~~
main
baz



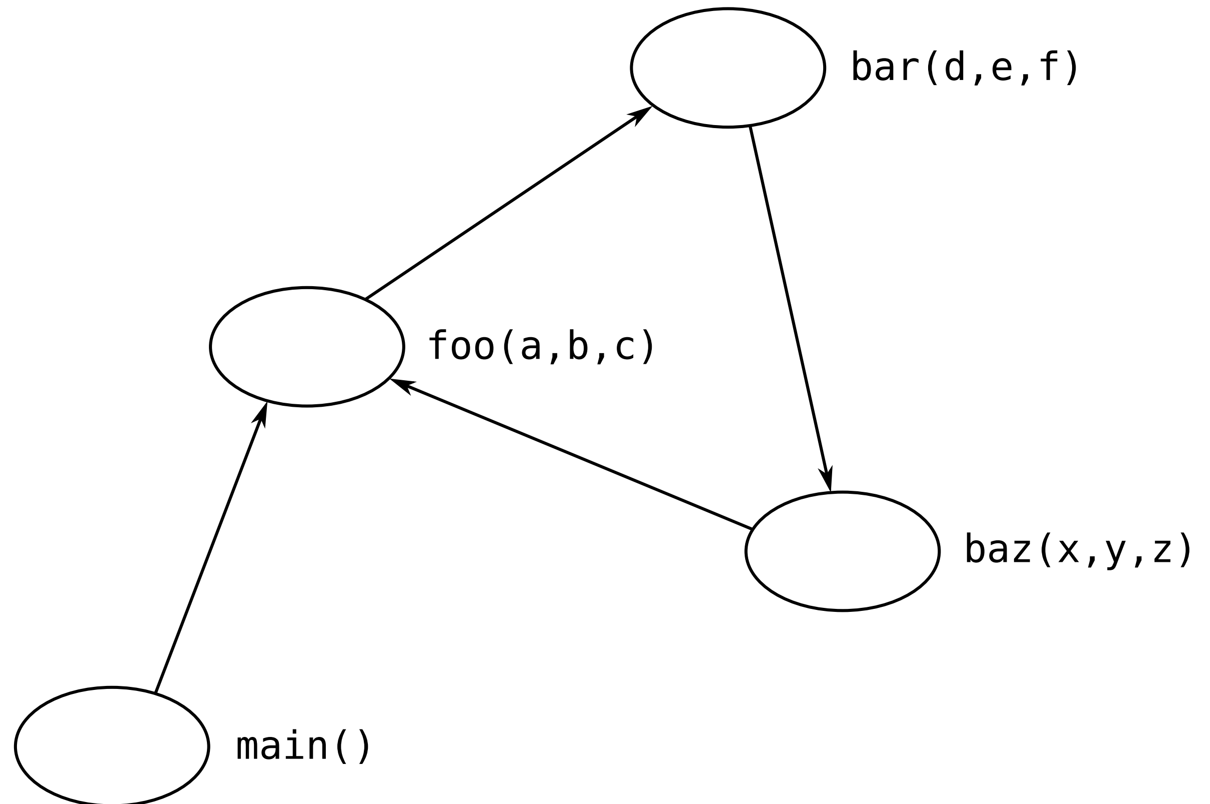
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

- ~~baz~~
- ~~bar~~
- ~~foo~~
- ~~main~~
- baz



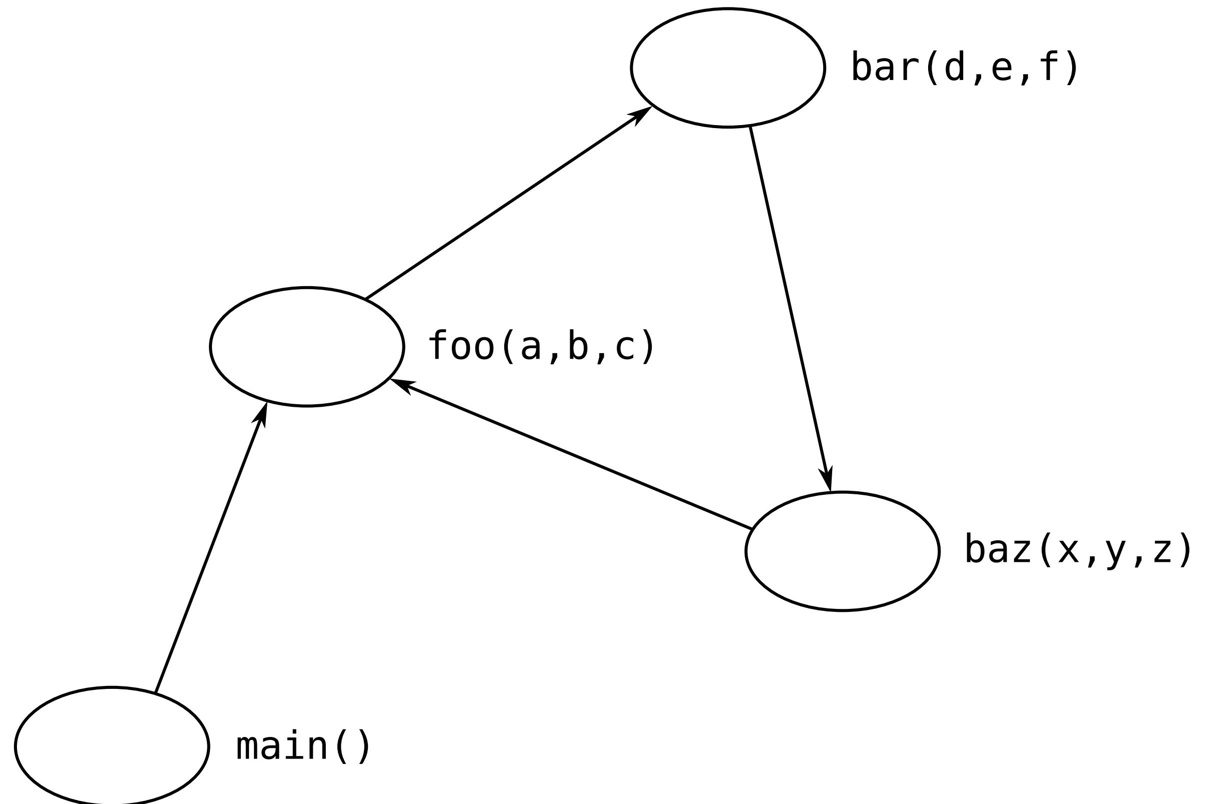
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

~~baz~~
~~bar~~
~~foo~~
~~main~~
~~baz~~
bar



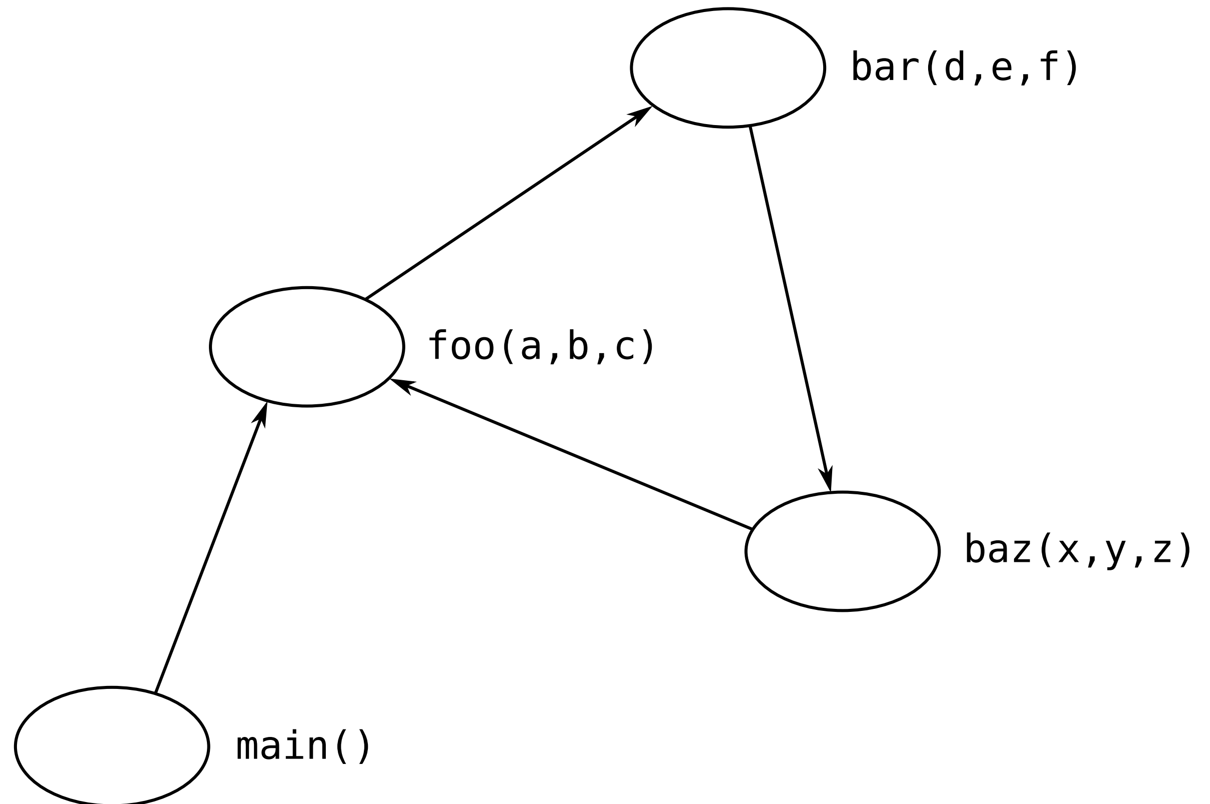
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

~~baz~~
~~bar~~
~~foo~~
~~main~~
~~baz~~
~~bar~~
foo



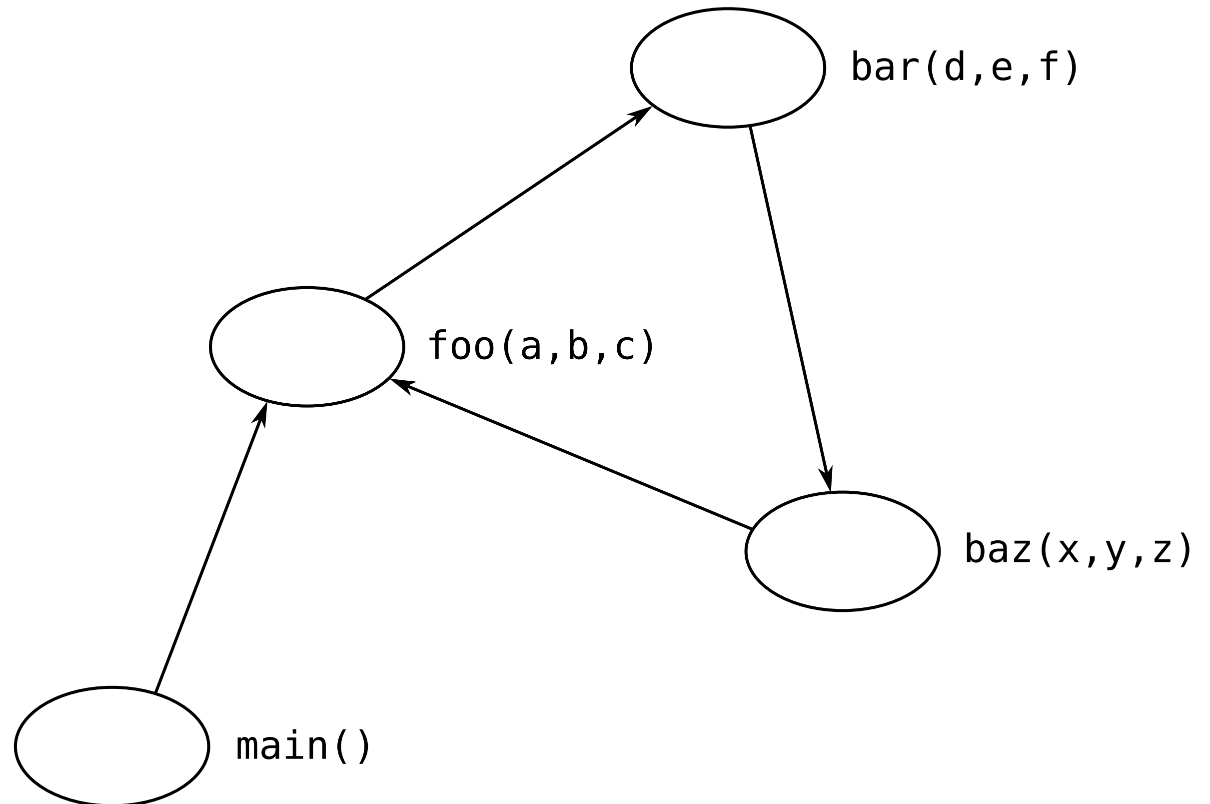
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

~~baz~~
~~bar~~
~~foo~~
~~main~~
~~baz~~
~~bar~~
~~foo~~
main
baz



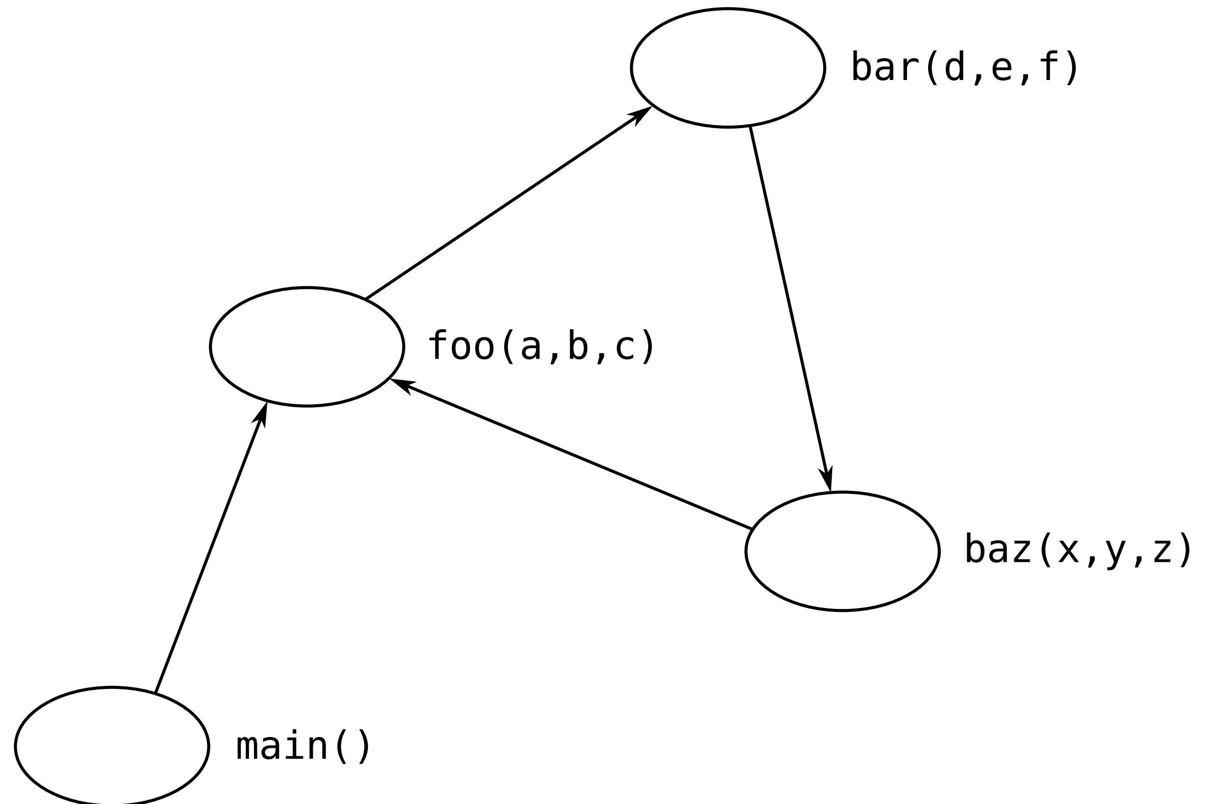
Interprocedural Parameter Dependence

```
r = foo(a,b,c)
```

which uses do we add?

worklist

~~baz~~
~~bar~~
~~foo~~
~~main~~
~~baz~~
~~bar~~
~~foo~~
main
baz
•
•
•



Parameter Dependence Results

dependence type	consumed callsites	non-void methods
full	51%	58%
partial	22%	9%
zero, with parameters	24%	30%
zero, without parameters	3%	3%

- Memoization at 22% of callsites will be improved
- 27% of callsites do not benefit from memoization
- Average taken over SPEC JVM98

Return Value Use Analysis

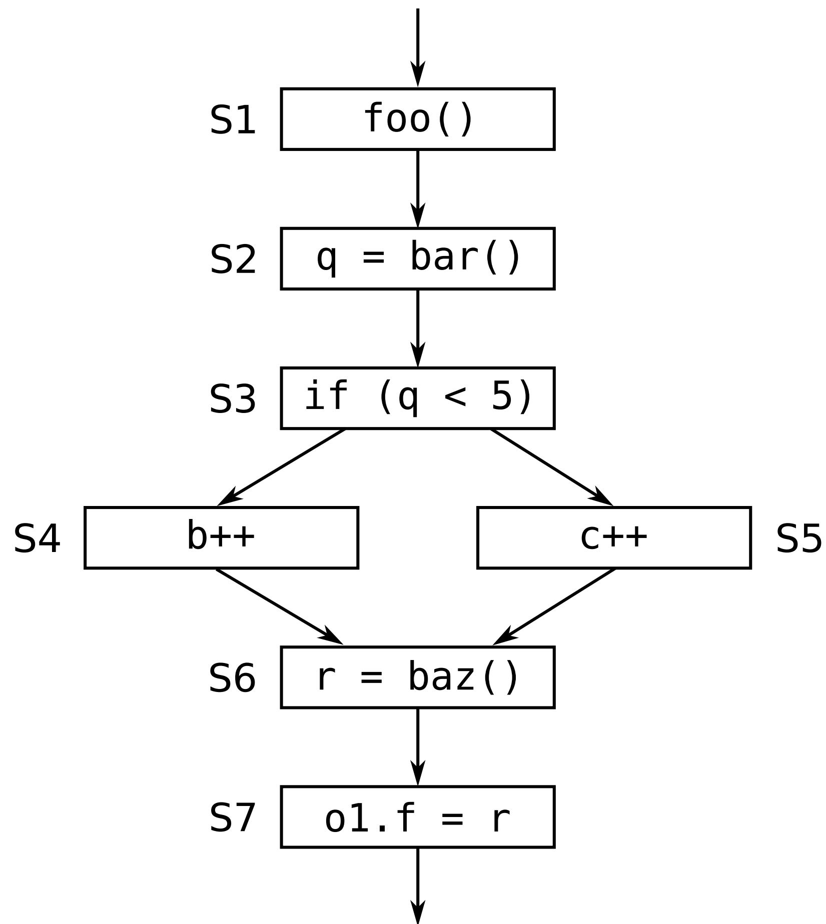
- An incorrect return value *r* may be OK
  - If *r* is unused
  - If *r* appears inside a boolean expression

```
r = foo (a, b, c);
if (r > 10)
   {
      ...       // r == 11, 12, 13, ...
   }
else
   {
      ...       // r == 10, 9, 8, ...
   }
```

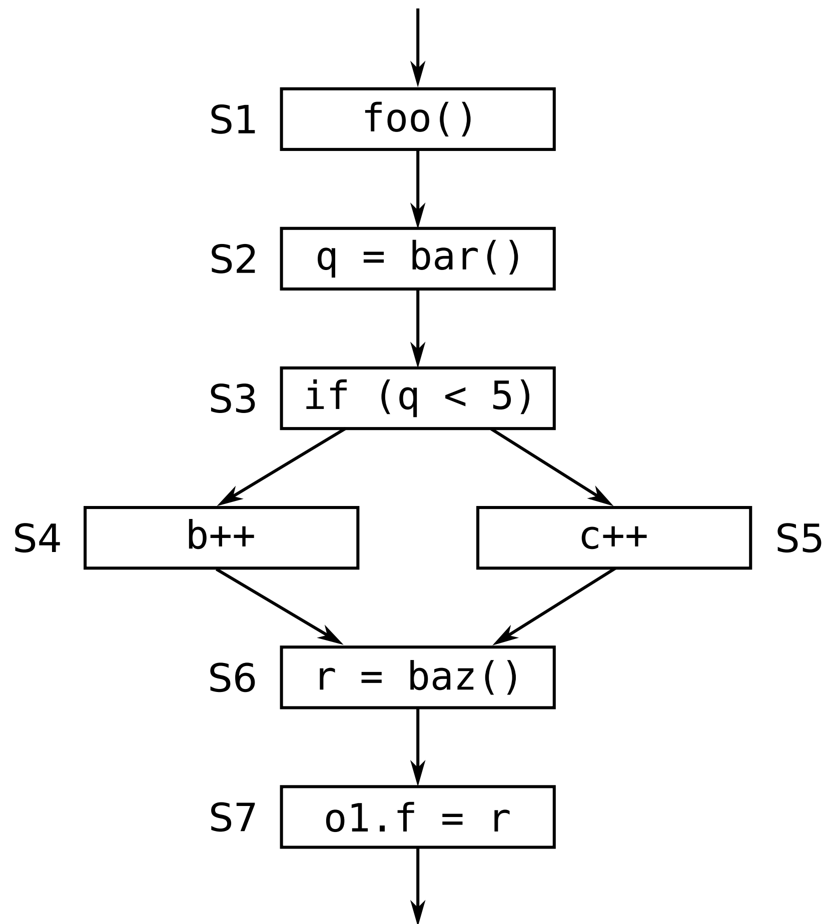
Return Value Use Analysis

- Collect *use expressions* for each return value
- Evaluate use expressions at runtime
  - If predicted and actual return values satisfy use expressions identically, we can substitute an inaccurate prediction
    - ++accuracy

Return Value Use

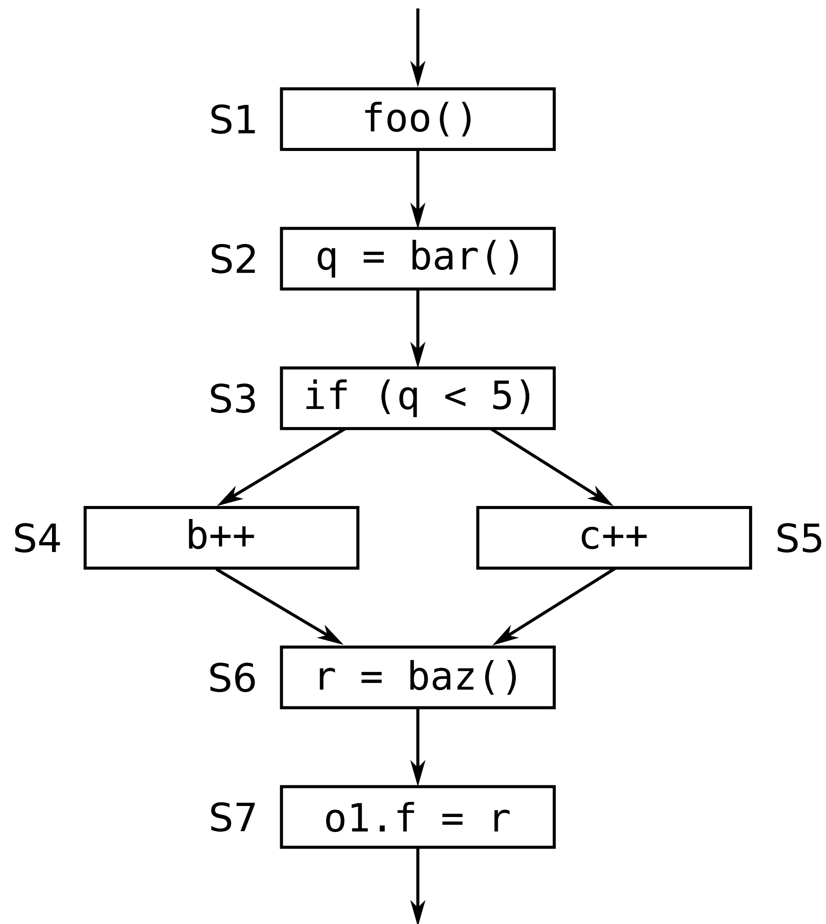


Return Value Use



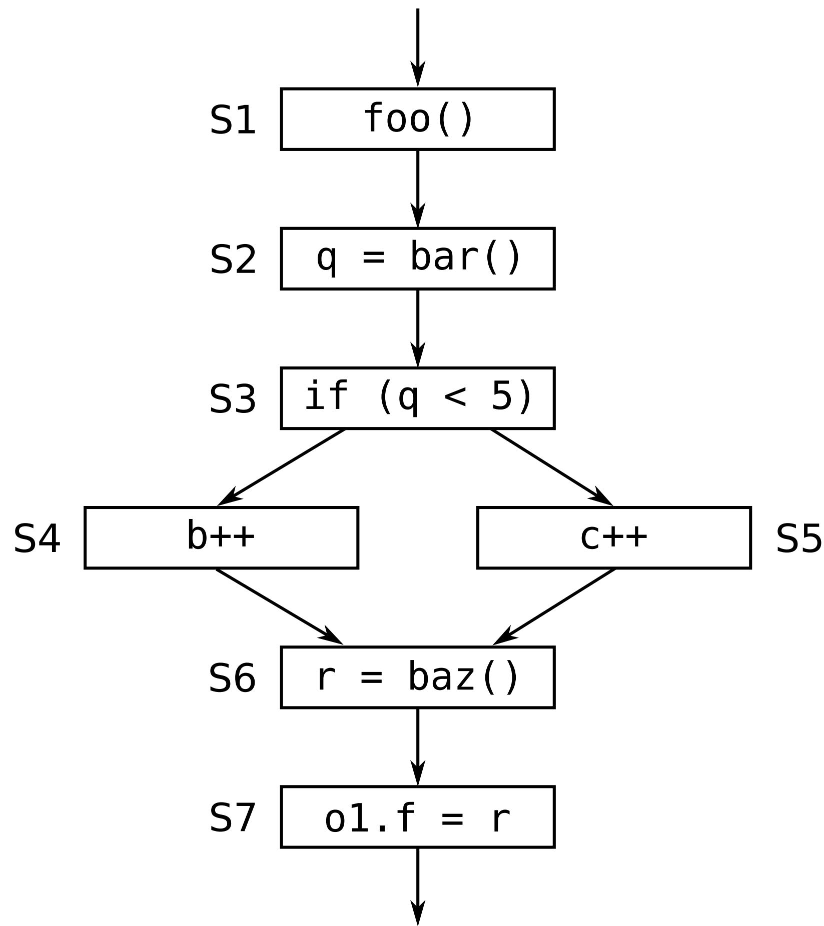
	consumed	accurate	uses
S1			
S2			
S6			

Return Value Use



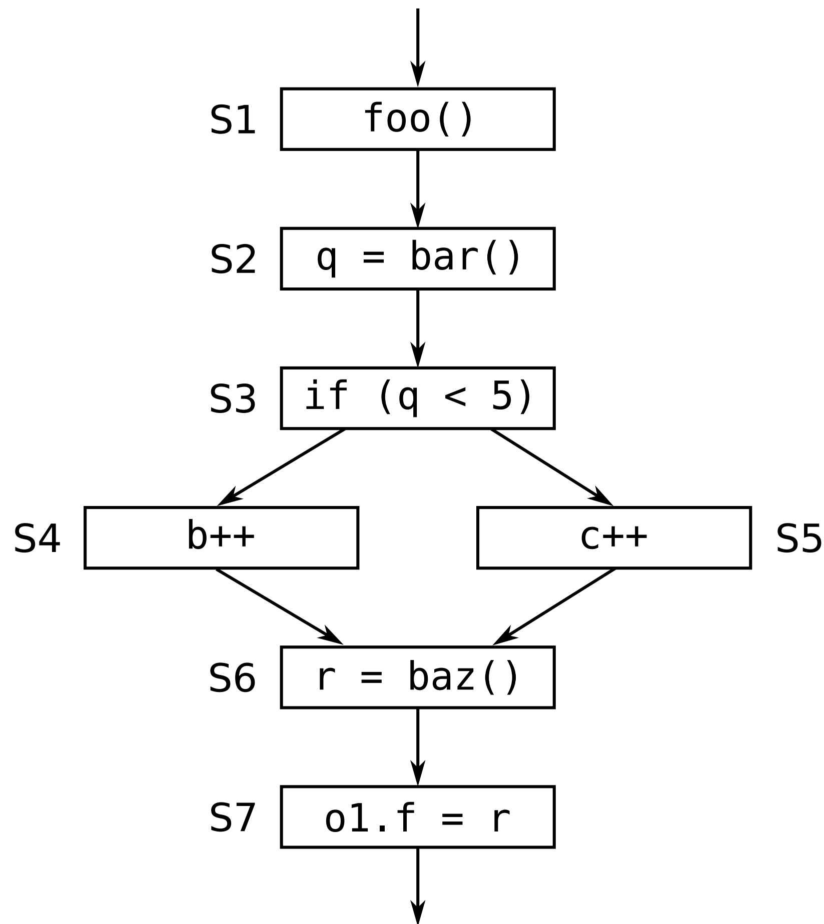
	consumed	accurate	uses
S1	no	no	———
S2			
S6			

Return Value Use



	consumed	accurate	uses
S1	no	no	———
S2	yes	no	q < 5
S6			

Return Value Use



	consumed	accurate	uses
S1	no	no	———
S2	yes	no	q < 5
S6	yes	yes	———

Return Value Use Results

consumed	accurate	callsites (%)
no	no	21
yes	no	10
yes	yes	69

- Use expressions only involve r and a constant
- Future: allow for locals as well as constants
  - Relax accuracy constraints further

Conclusions (1)

- Two new compiler analyses for improved RVP
  - Parameter dependence analysis: *production*
  - Return value use analysis: *consumption*
- Static results look promising:
  - 22% of callsites have partial dependencies
  - 27% of callsites have zero dependencies
  - 21% of return values are unconsumed
  - 31% of return values may be inaccurate

Conclusions (2)

- Parameter dependence analysis is optimistic
  - Conservative correctness not required
- Return value use analysis relaxes safety constraints

Future Work

- Allow for comparisons with locals in use expressions. At runtime, these values may be:
  - Parameter locals
  - Non-parameter locals
  - Stack values
- Determine effect of analyses at runtime
- Implement purity analysis in Soot (Sălcianu, Rinard)
  - Skip pure methods altogether via memoization!
- Finish SMLP implementation in SableVM
  - Study costs and benefits of RVP in this system