Static Analysis with Soot

SMA HS 2018
Manuel Leuenberger
PhD student at SCG
leuenberger@inf.unibe.ch

Soot is a static analysis framework

- originally an optimization framework (used in compilers)
- understands JVM languages (Java, Android, etc.)
- whole-program analysis (call graph construction)
- dataflow analysis (nullness, array boundary checks)

…but first some theory

Dataflow analysis

- dataflow analysis is a form of abstract interpretation, i.e. reason about some properties of the program state at a certain block
- different types:
  - forward (reaching definitions)
  - backward (liveness)
  - branched (nullness)

Reaching definitions

```
int gcd(int a, int b) {
  int c = a;
  int d = b;
  if (c == 0) {
    return d;
  }
  while (d != 0) {
    if (c > d) {
      c = c - d;
    } else {
      d = d - c;
    }
  }
  return c;
}
```

which definitions reach here?

Reaching definitions

- *Which definitions reach a block?*
- used in compiler optimizations
  - constant folding
  - common subexpression elimination
  - use-def/def-use chains

Potential optimizations

```
int gcd(int a, int b) {
  int c = a;
  int d = b;
  if (c == 0) {
    return d;
  }
  while (d != 0) {
    if (c > d) {
      c = c - d;
    } else {
      d = d - c;
    }
  }
  return c;
}
```

c = a, parameter a not changed → a == 0

Potential optimizations

```
int gcd(int a, int b) {
   int c = a;
   int d = b;
   if (a == 0) {
      return d;
   }
   while (d != 0) {
      if (c > d) {
         c = c - d;
      } else {
         d = d - c;
      }
   }
   return c;
}
```

d = b, parameter b not changed → return b

Potential optimizations

```c
int gcd(int a, int b) {
   int c = a;
   int d = b;
   if (a == 0) {
      return b;
   }
   while (d != 0) {
      if (c > d) {
         c = c - d;
      } else {
         d = d - c;
      }
   }
   return c;
}
```

c, d not used,
a, b unchanged
→ allocate later

Potential optimizations

```
int gcd(int a, int b) {
  if (a == 0) {
    return b;
  }
  int c = a;
  int d = b;
  while (d != 0) {
    if (c > d) {
      c = c - d;
    } else {
      d = d - c;
    }
  }
  return c;
}
```

a, b only used in definition
→ reuse registers

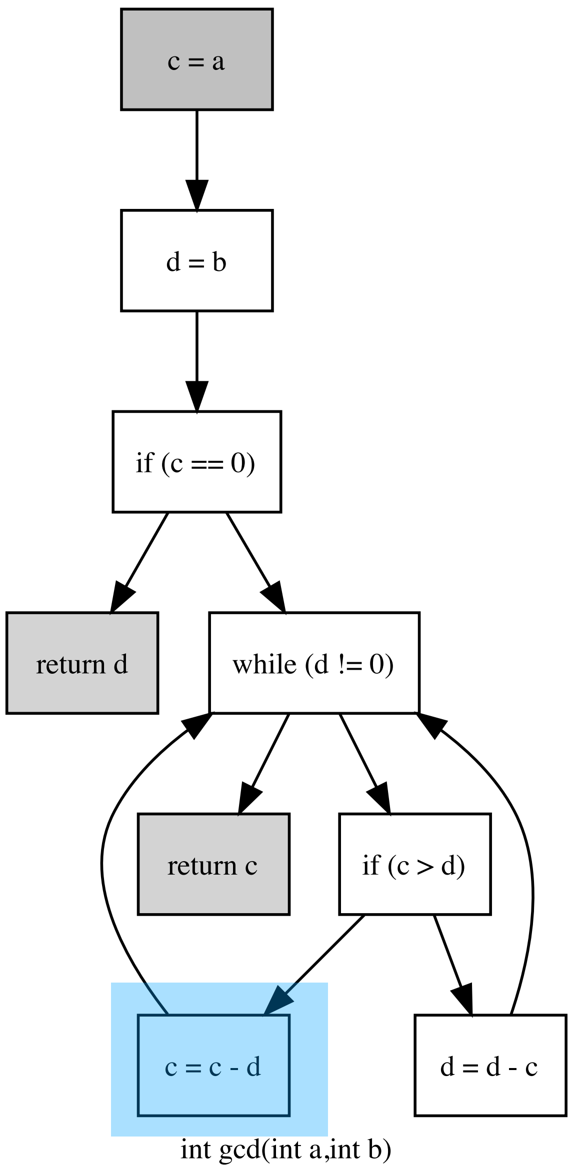
Potential optimizations

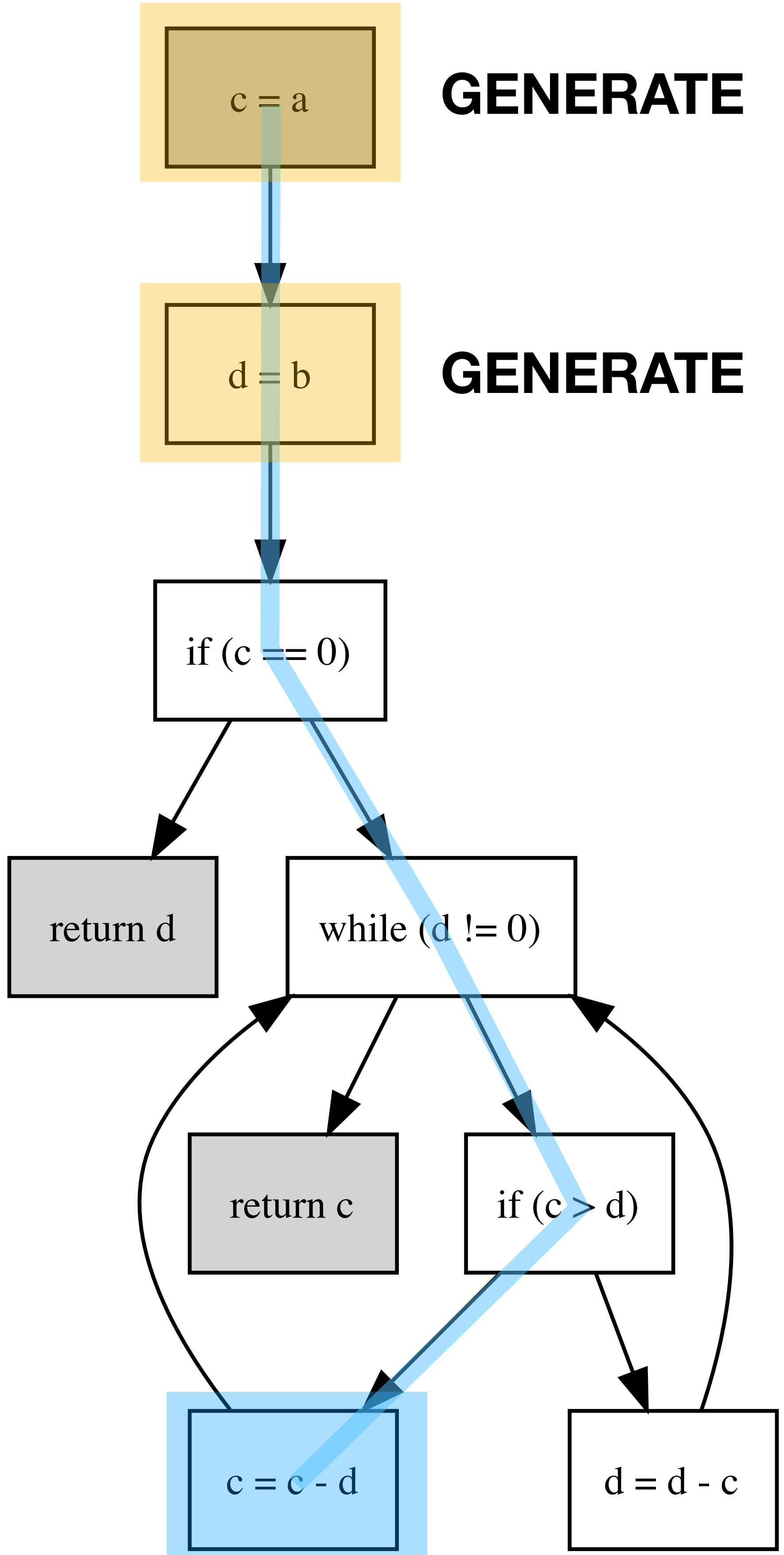
```c
int gcd(int a, int b) {
  if (a == 0) {
    return b;
  }
  while (b != 0) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }
  return a;
}
```

optimized register allocation!

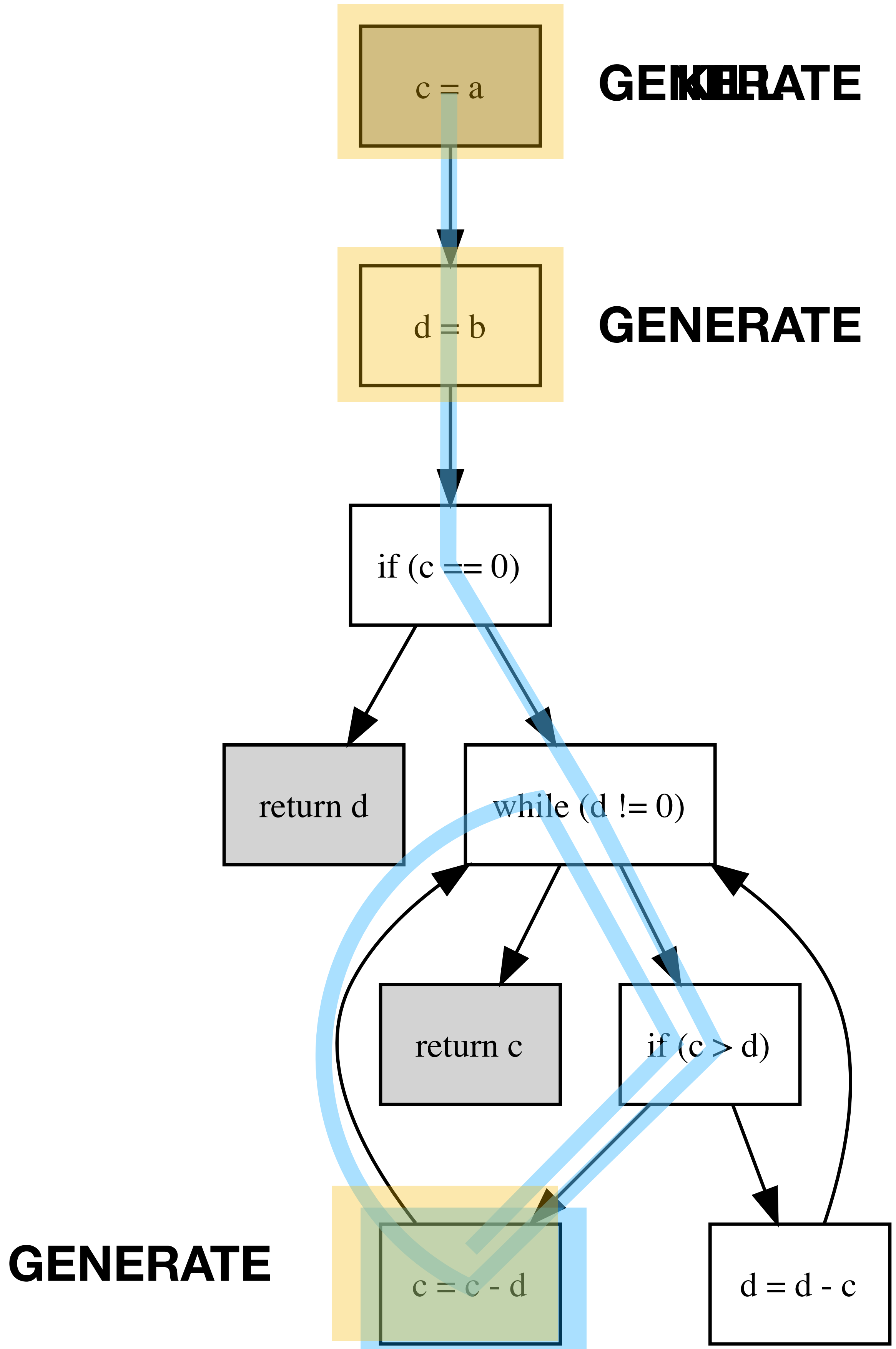
Control-flow graphs

```
int gcd(int a, int b) {
   int c = a;
   int d = b;
   if (c == 0) {
      return d;
   }
   while (d != 0) {
      if (c > d) {
         c = c - d;
      } else {
         d = d - c;
      }
   }
   return c;
}
```

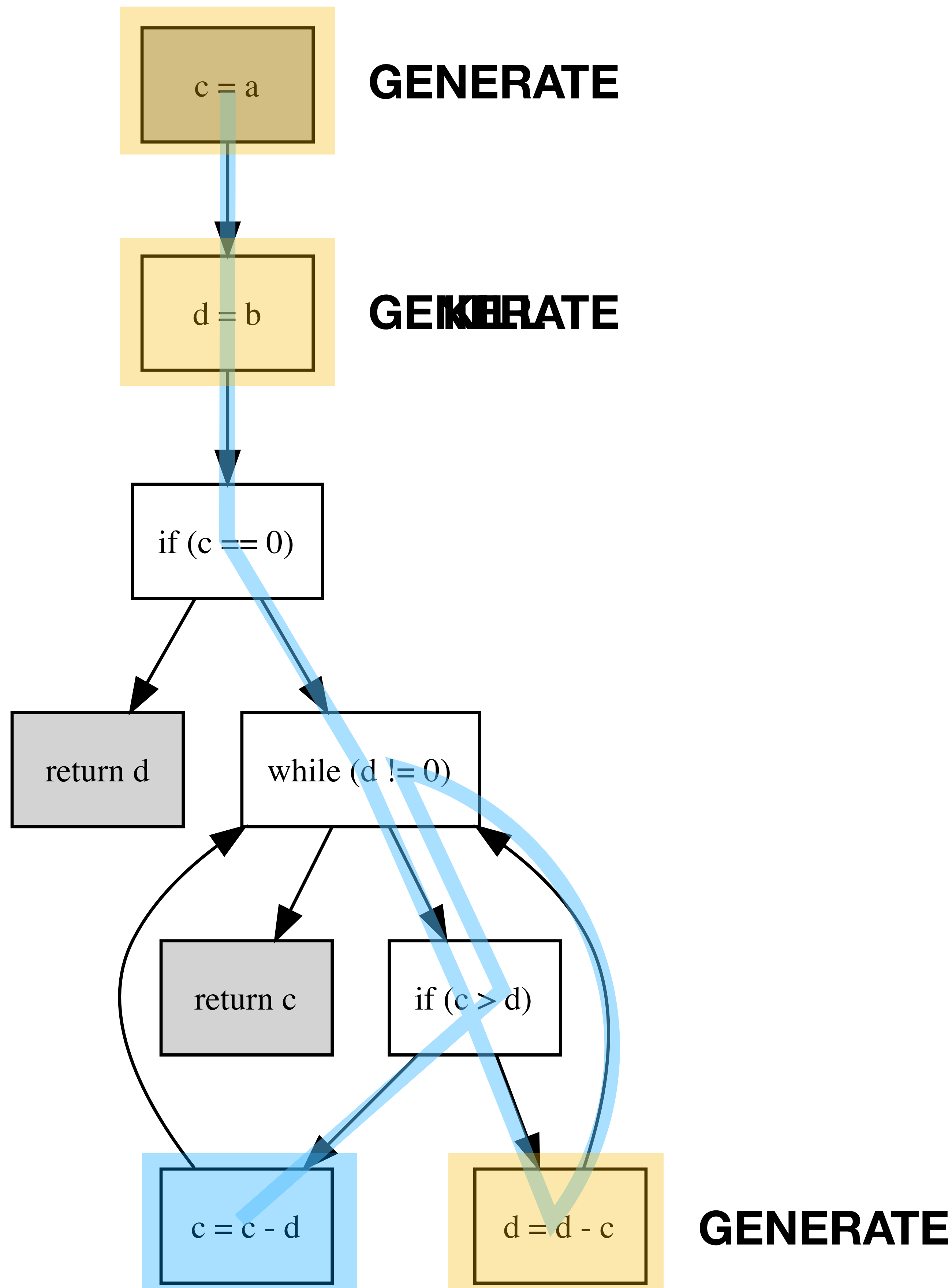




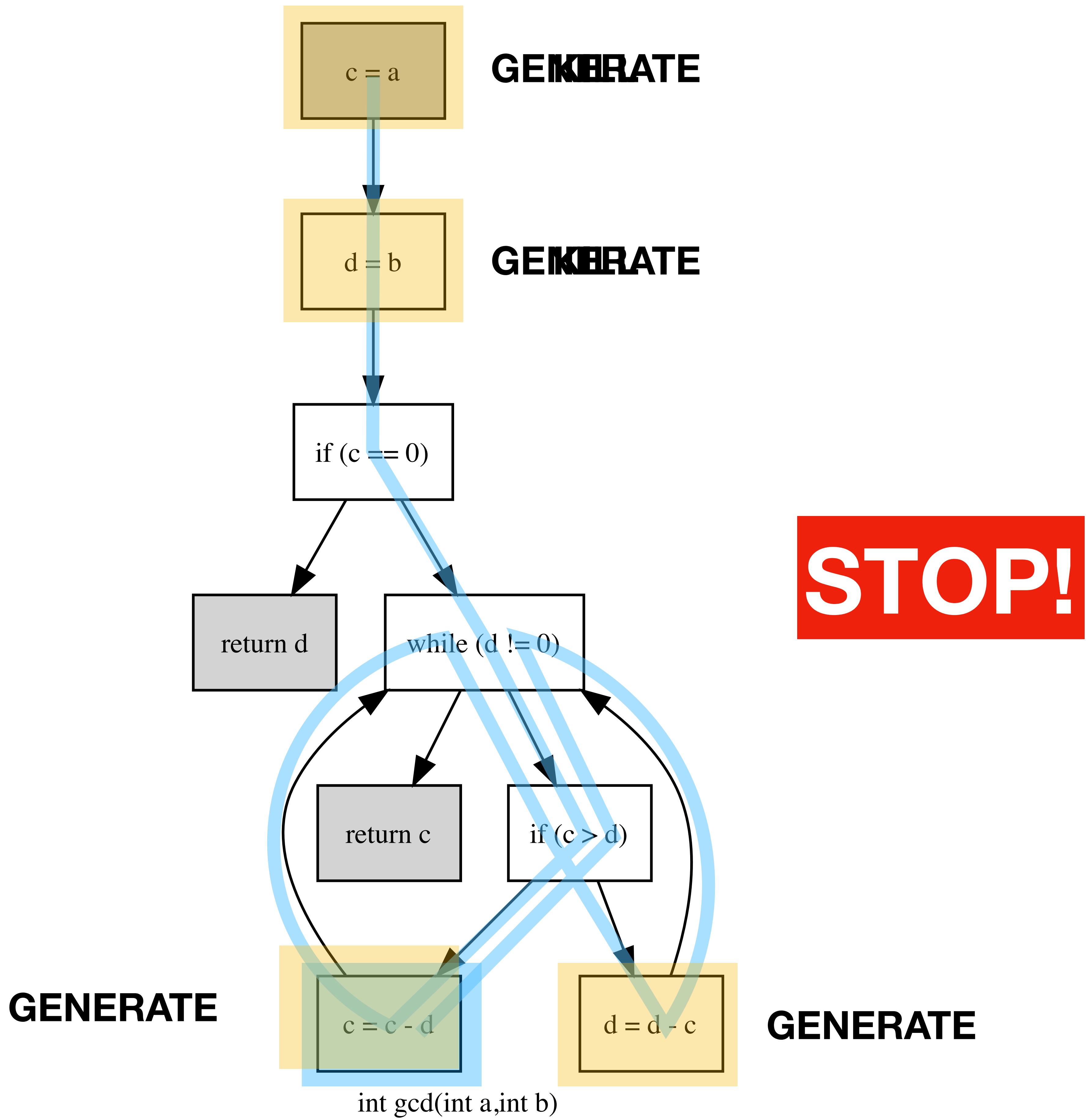
int gcd(int a,int b)



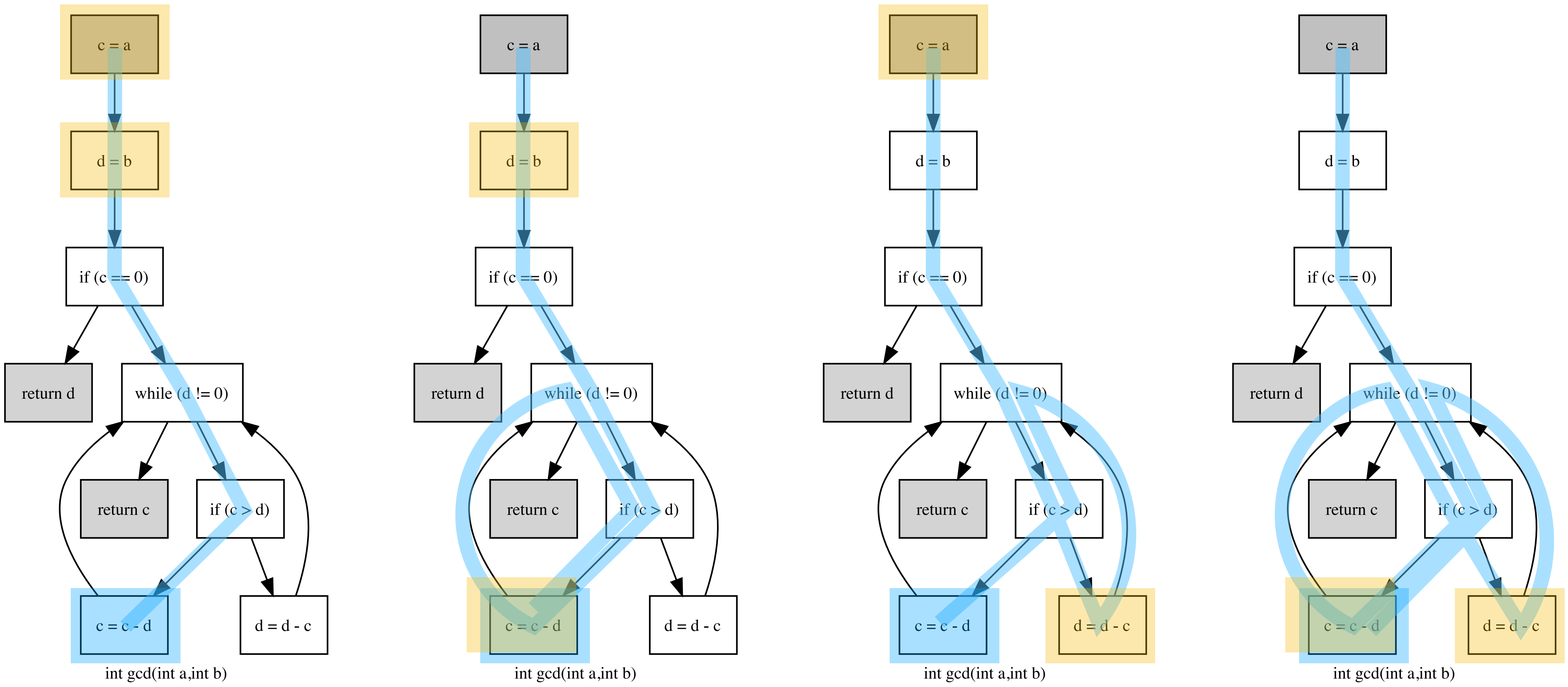
int gcd(int a,int b)



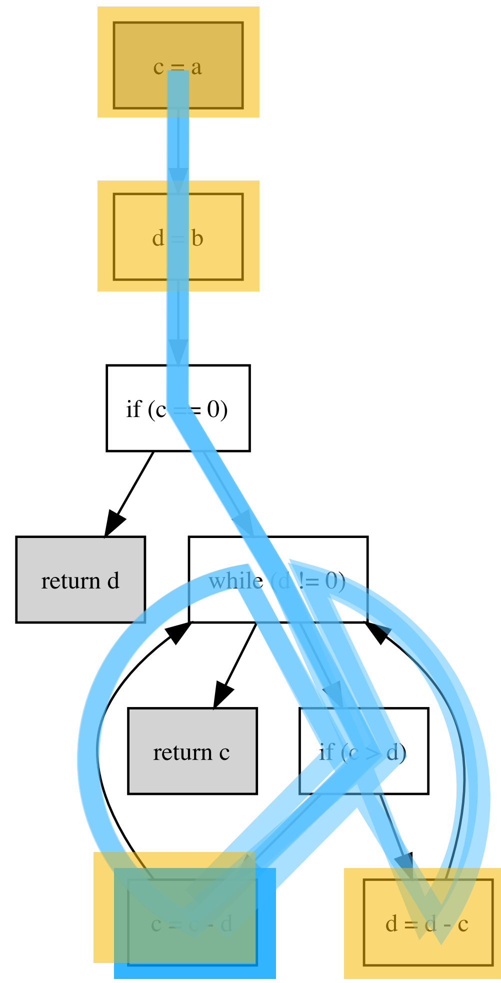
int gcd(int a,int b)

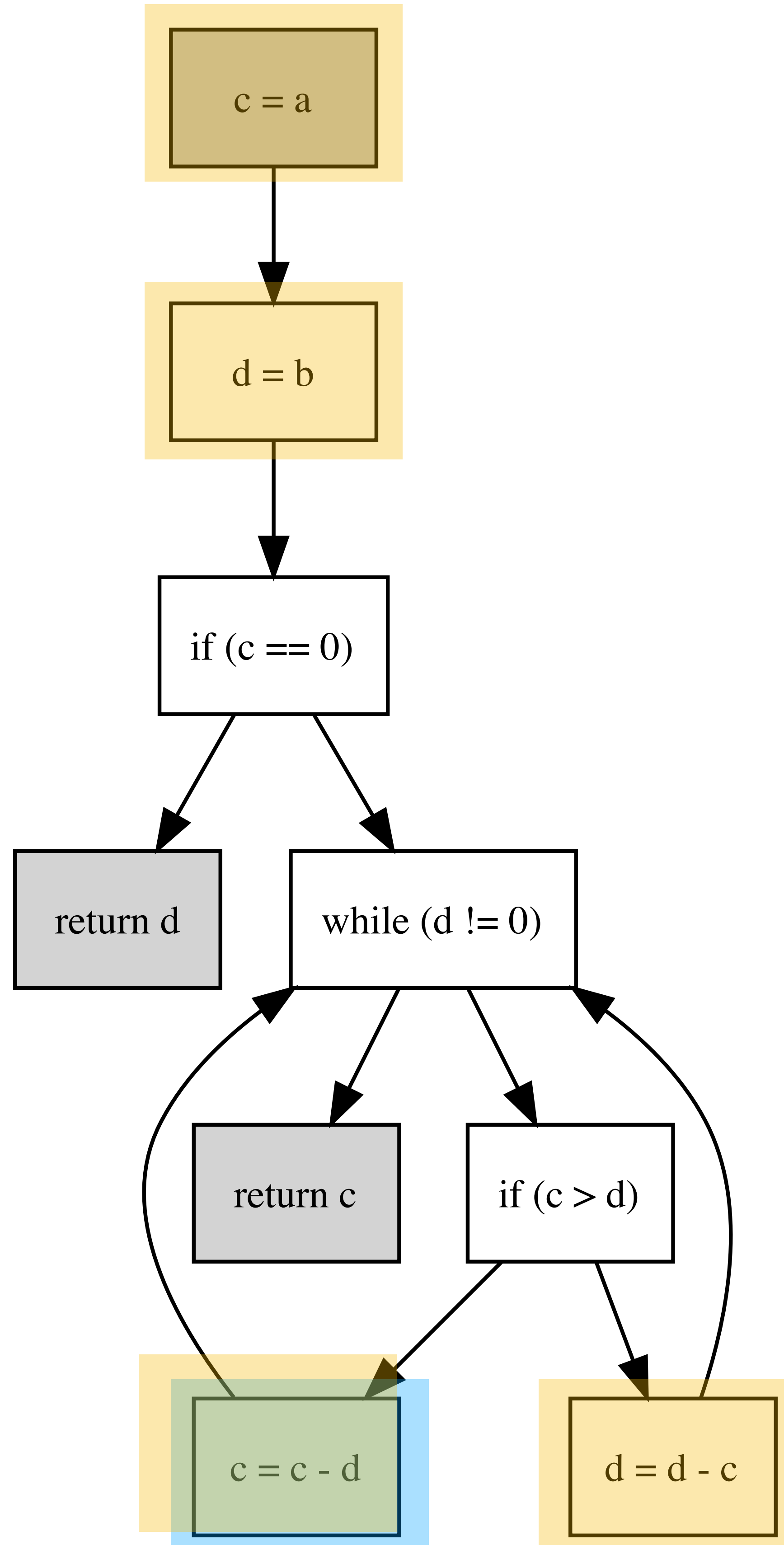

c = a
GENERATE
KILL
d = b
GENERATE
KILL
if (c == 0)
return d
while (d != 0)
STOP!
return c
if (c > d)
GENERATE
c = c - d
d = d - c
GENERATE
int gcd(int a,int b)

How to merge?


c = a
d = b
if (c == 0)
return d
while (d != 0)
return c
if (c > d)
c = c - d
d = d - c
int gcd(int a,int b)
c = a
d = b
if (c == 0)
return d
while (d != 0)
return c
if (c > d)
c = c - d
d = d - c
int gcd(int a,int b)
c = a
d = b
if (c == 0)
return d
while (d != 0)
return c
if (c > d)
c = c - d
d = d - c
int gcd(int a,int b)
c = a
d = b
if (c == 0)
return d
while (d != 0)
return c
if (c > d)
c = c - d
d = d - c
int gcd(int a,int b)

Union!


c = a
d = b
if (c == 0)
return d
while (d != 0)
return c
if (c > d)
c = c - d
d = d - c
int gcd(int a,int b)



int gcd(int a,int b)

Dataflow equation

$$\text{REACH}_{\text{in}}[S] = \bigcup_{p \in pred[S]} \text{REACH}_{\text{out}}[p]$$

all definitions reaching after all predecessors

c = c - d

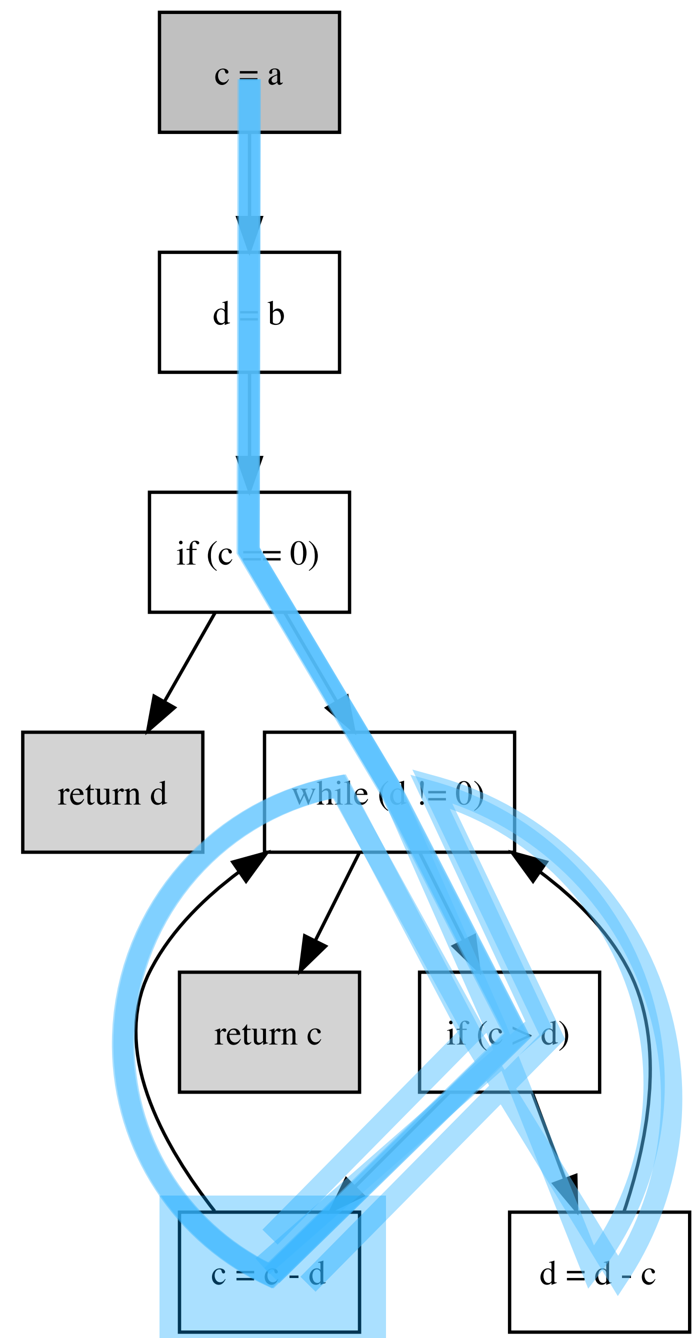
c = c - d

all definitions of c

$$\text{REACH}_{\text{out}}[S] = \text{GEN}[S] \cup (\text{REACH}_{\text{in}}[S] - \text{KILL}[S])$$

Dataflow equation solver

- iterative round-robin
- start with initial (empty) set for each block input
- compute output of each block whenever its input (=output of predecessors) change
- repeat until no outputs change



int gcd(int a,int b)

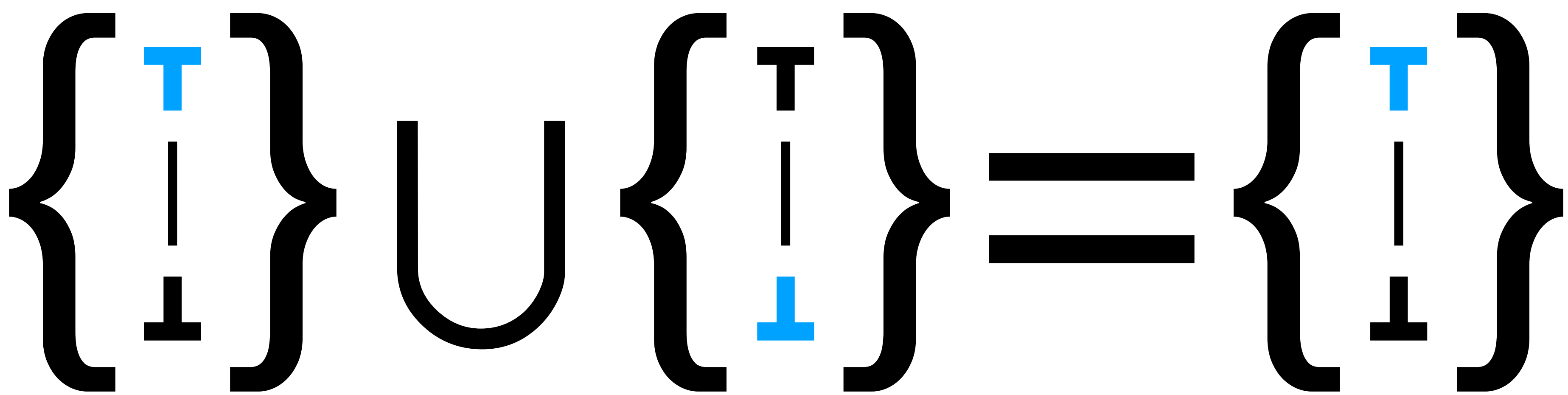
Ensure convergence

- a block output needs to reach a fix-point…
- …or the solver never terminates
- “unify towards top of lattice”

$$\mathrm{REACH}_{\mathrm{in}}[S] = \bigcup_{p \in pred[S]} \mathrm{REACH}_{\mathrm{out}}[p]$$

lattice for each element in REACH, e.g., `d = d - c` :

lattice top: “reaches block”
lattice bottom: “does not reach block”



REACH(out, p1) REACH(out, p2) REACH(in, S)

Enough theory!
Where is the code?

Soon…

Java is complex

```
c = a > 5 ? b : d;
```

Jimple is simple

```
int gcd(int, int)
{
    ch.unibe.scg.sma.soot.TestClass this;
    int a, b, c, d;
    this := @this: ch.unibe.scg.sma.soot.TestClass;
    a := @parameter0: int;
    b := @parameter1: int;
    c = a;
    d = b;
    if a != 0 goto label3;
    return b;
 label1:
    if c <= d goto label2;
    c = c - d;
    goto label3;
 label2:
    d = d - c;
 label3:
    if d != 0 goto label1;
    return c;
}
```

```
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (c == 0) {
        return d;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```

```
int gcd(int, int)
{
    ch.unibe.scg.sma.soot.TestClass this;
    int a, b, c, d;
    this := @this: ch.unibe.scg.sma.soot.TestClass;
    a := @parameter0: int;
    b := @parameter1: int;
    c = a;
    d = b;
    if a != 0 goto label3;
    return b;
 label1:
    if c <= d goto label2;
    c = c - d;
    goto label3;
 label2:
    d = d - c;
 label3:
    if d != 0 goto label1;
    return c;
}
```

mostly only one thing per statement

already optimized!

branches…

…and jumps

```
int gcd(int a, int b) {
    int c = a;
    int d = b;
    if (c == 0) {
        return d;
    }
    while (d != 0) {
        if (c > d) {
            c = c - d;
        } else {
            d = d - c;
        }
    }
    return c;
}
```

A simple IR is crucial!

- small instruction set, i.e. just a few statement types
- statements do one thing only
- flat structure
- one scope per method, no nesting
- many special cases in Java are common cases in Jimple

```
int fib(int)
{
    X this;
    int n, $i0, $i1, $i2, $i3, $i4;
    this := @this: TestClass;
    n := @parameter0: int;
    if n > 1 goto label1;
    return n;
 label1:
    $i0 = n - 2;
    $i1 = virtualinvoke this.<X: int fib(int)>($i0);
    $i2 = n - 1;
    $i3 = virtualinvoke this.<X: int fib(int)>($i2);
    $i4 = $i1 + $i3;
    return $i4;
}
```

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 2) + fib(n - 1);
}
```

Reaching Definitions

- intra-procedural: analyze each method independently
- only look at definitions of locals
- forward flow: knowledge flows along control-flow

$$\mathrm{REACH_{in}}[S] = \bigcup_{p \in pred[S]} \mathrm{REACH_{out}}[p]$$

$$\mathrm{REACH_{out}}[S] = \mathrm{GEN}[S] \cup (\mathrm{REACH_{in}}[S] - \mathrm{KILL}[S])$$

Code!

Other applications

- Nullness analysis
  - Which values can be null?
  - What is checked for null?
- Protocols
  - How are objects initialized?
  - Which methods do I need to call before doing X?
- Inter-procedural analysis
  - Sinks and sources: Can confidential information leak?